Git
Fundamentos



Git. Fundamentos

Desarrollado por Jesús Amieiro Becerra

Versión actual: 27/10/2015

Puedes obtener la última versión disponible en http://fontelearn.com/es/git-fundamentos/ o en http://www.jesusamieiro.com/docs/

Índice de contenido

1 Presentación

Este es el primero de una serie de libros sobre Git, un sistema de control de versiones desarrollado por Linux Torvalds en el año 2005 y que se ha hecho tremendamente popular gracias a servicios como GitHub y a su amplia aceptación en proyectos importantes como el Kernel de Linux, Android, Ruby on Rails, Eclipse, GNOME, KDE, Qt, Perl o PostgreSQL o por empresas como Google, Facebook, Microsoft, Twitter, LinkedIn o Netflix.

Si eres programador, desarrollador web, administrador de sistemas, diseñador, ... es muy probable que en algún momento de tu trabajo te encuentres con un proyecto en el que tengas que colaborar con otras personas usando Git. Puede que trabajes solo pero que te interese tener un seguimiento y control de tu trabajo. En estos dos casos y en muchos más un conocimiento más o menos profundo de Git te permitirá ser mucho más productivo en tu trabajo y, sobre todo, evitar muchos de los problemas con los que se encuentra a menudo la gente que no trabaja con un sistema de control de versiones.

Si tu ámbito de trabajo es técnico y aún no usas Git, cuando lleves unos meses usándolo te preguntarás cómo es posible que no lo hubieras empezado a usar antes.

1.1 Contenido

Este libro está en versión **alfa**. Esto quiere decir que le faltan contenidos y, aunque hemos tratado de evitar cualquier tipo de error, puede que encuentres alguno. En ese caso te agradecería que lo notificaras a través de cualquiera de los métodos de contacto indicados en el capítulo *1.2 Contacto*.

¿Qué vas a encontrar en este libro?

En el capítulo *2 Introducción* vamos a presentar los sistemas de control de versiones, qué son, qué problemática solucionan y quiénes deberían usar un sistema de control de versiones. Veremos los distintos tipos de sistemas de control de versiones atendiendo a su arquitectura (local, centralizado y distribuido), realizaremos una breve reseña histórica de cómo se creó Git y cuáles son sus características fundamentales.

En el capítulo *3 Instalación* vamos a explicar brevemente cómo se instala Git en los 3 sistemas operativos más utilizados actualmente: Windows, Linux (Debian y Fedora) y OS X.

En el capítulo *4 Primeros pasos* veremos cómo obtener ayuda de Git, cómo se configura Git por primera vez e introduciremos una serie de conceptos básicos pero fundamentales para trabajar con Git: repositorio, commit, las zonas en Git, los estados de un archivo, el flujo de trabajo básico, SHA-1, HEAD y rama. A continuación veremos cómo se inicializa un repositorio, cómo se añaden archivos al repositorio, cómo se realizan los commits, cómo se gestionan los cambios en los archivos, cómo se renombran archivos, cómo se borran archivos y cómo se ignoran archivos que no queremos controlar con Git.

En el capítulo *5 Ramas* veremos qué son las ramas, por qué usarlas, cómo crearlas, cómo ver las que tenemos, cómo cambiar de rama, cómo renombrar una rama, cómo borrar una rama, cómo fusionar el contenido de dos ramas y cómo arreglar conflictos entre su contenido. También veremos con funciona el rebase y el stash.

En el capítulo 6 Historial veremos cómo poder filtrar y realizar búsquedas en los distintos elementos de un proyecto.

En el capítulo 7 Cambios en el proyecto veremos cómo poder filtrar y visualizar los cambios que van teniendo lugar a lo largo del transcurso del proyecto.

En el capítulo 13 Repositorios remotos veremos cómo podemos colaborar con otros usuarios a través de un repositorio remoto. En este caso usaremos BitBucket, pero también podríamos haber

usado otros bastante conocidos como son GitHub o GitLab.

1.2 Contacto

Puedes ponerte en contacto con nosotros a través de cualquiera de los métodos de contacto indicados en la dirección web http://fontelearn.com/es/contacto/

2 Introducción

2.1 ¿Qué es un sistema de control de versiones?

Git es un sistema de control de versiones diseñado para la gestión de proyectos, enfocado principalmente a los proyectos de desarrollo de software.

Pero, ¿qué es un control de versiones? Pues simplemente es un sistema que registra los cambios que se producen en un conjunto de archivos que forman parte de un proyecto a lo largo del tiempo, de tal forma que en cualquier momento podemos regresar a una versión antigua, ver los cambios que hemos realizado en un determinado archivo o quién lo ha hecho, entre otras muchas funcionalidades que veremos a lo largo del libro.

Git fue diseñado para gestionar de una forma eficiente los cambios que tienen lugar en los archivos, principalmente en los de texto.

Esto no quiere decir que si somos diseñadores gráficos o trabajamos con archivos de vídeo no debamos usar Git; es más, es muy aconsejable, ya que podremos volver a una versión anterior de uno o de varios archivos, ver quién ha realizado determinados cambios que han introducido un problema o recuperar archivos que creíamos que habían sido borrados.

Habitualmente, cuando trabajamos tenemos la necesidad de gestionar distintas versiones de archivos para poder controlar los cambios que se producen a lo largo del tiempo.

Existen muchas formas de hacerlo, como por ejemplo numerando los archivos con el número de la versión.

- Presupuesto_v2.doc
- Cartel_v5.jpg

En estos dos ejemplos podemos ver la segunda versión de un presupuesto o la quinta versión de un cartel.

También es habitual guardar una copia de todos los archivos de un proyecto en un directorio con la fecha de copia.

- 2010_05_17_web

Estos métodos se utilizan porque son muy cómodos pero también muy **propensos a errores**, tales como la edición de una versión incorrecta de un archivo o sobrescribir un directorio que no deberíamos machacar.

También existen programas informáticos que implementan un control de cambios. Un ejemplo lo podemos encontrar en Microsoft Word o en LibreOffice Writer, que permiten llevar a cabo un control y seguimiento de los cambios.

También existen herramientas web, como pueden ser las wikis, cuyo ejemplo más conocido es la Wikipedia.

Cuando necesitamos gestionar los distintos archivos de un proyecto, como por ejemplo el código fuente de un software o los archivos HTML, CSS y Javascript de una página web estática, utilizamos un sistema de control de versiones, conocido en inglés como ***Version Control System (VCS)*** o ***Source Control Management (SCM)***, ya que nos permite registrar los cambios que se producen en una serie de archivos en el tiempo.

Todo ello lo hacemos utilizando con elemento fundamental el **commit**, que viene a ser una instantánea del proyecto en un momento determinado, que se guarda en la base de datos del VCS. Utilizando la información guardada a lo largo del tiempo a través de los commits, un sistema de control de versiones nos permite:

- Revertir uno o varios archivos a un estado previo.
- Comparar los cambios que se han producido en determinados archivos.
- Poder ver quién realizó un determinado cambio.

Y, en general, realizar un control exhaustivo de los cambios y poder llevar el proyecto a cualquier otro estado anterior.

2.2 ¿Quién debe usar un control de versiones?

Git fue diseñado para gestionar de una forma eficiente los cambios que tienen lugar en los archivos, principalmente en los de texto, ya que su objetivo principal es la gestión de proyectos de software, en los que la mayoría de archivos son texto plano. No importa el tipo de lenguaje en el que desarrolles tu trabajo diario:

- Lenguajes interpretados: HTML, CSS, JavaScript, PHP, Perl, Python, Bash, ...
- Lenguajes compilados: C, C++, Java, VB.NET, C#, Objective-C, ...

Ya que podrás gestionar tu trabajo con Git.

Entonces, si eres diseñador gráfico, si trabajas con archivos de vídeo, retoque fotográfico, música, fuentes tipográficas, … ¿Git es para ti? Sí, y es muy aconsejable que lo uses, ya que te permite guardar las distintas versiones de los documentos y podrás volver a una versión anterior de uno o de varios archivos o recuperar archivos que creías que habían sido borrados. Estos archivos, al igual que otros como PDF, .docx, .odt, ..., que tienen que ser interpretados por una aplicación, para Git son binarios, de tal forma que no conoce las diferencias entre distintas versiones; es decir, no sabe que en la línea 24 se han cambiado dos palabras.

Lo que tenemos que tener claro es que toda la potencia de Git solo la vamos a obtener con archivos de texto plano, pero es muy recomendable usar Git en cualquier tipo de proyecto en el que nos interese tener acceso a un histórico de versiones, tanto si vamos a colaborar con más personas como si lo vamos a hacer solos.

2.3 Tipos de sistemas de control de versiones

Como hemos visto, los sistemas de control de versiones nos permiten registrar los cambios realizados a lo largo del tiempo en los archivos que pertenecen a un proyecto.

Posteriormente podemos ver los cambios que se han introducido, quién los ha introducido, podemos volver a una versión anterior de un archivo o de todo el proyecto, podemos comparar dos versiones distintas del proyecto, … y dispondremos de muchas otras funcionalidades que veremos a lo largo de los libros de Git.

A continuación vamos a ver los distintos tipos de control de versiones existentes, atendiendo a su arquitectura.

2.3.1 Control de versiones local

El control de versiones no es una tecnología reciente. Los primeros sistemas de control de versiones que aparecieron, los locales, permitían gestionar los cambios realizados en archivos locales.

El primer VCS como tal fue *Source Code Control System* o **SCCS**, que apareció en el año 1972 y que gozó de un gran éxito.

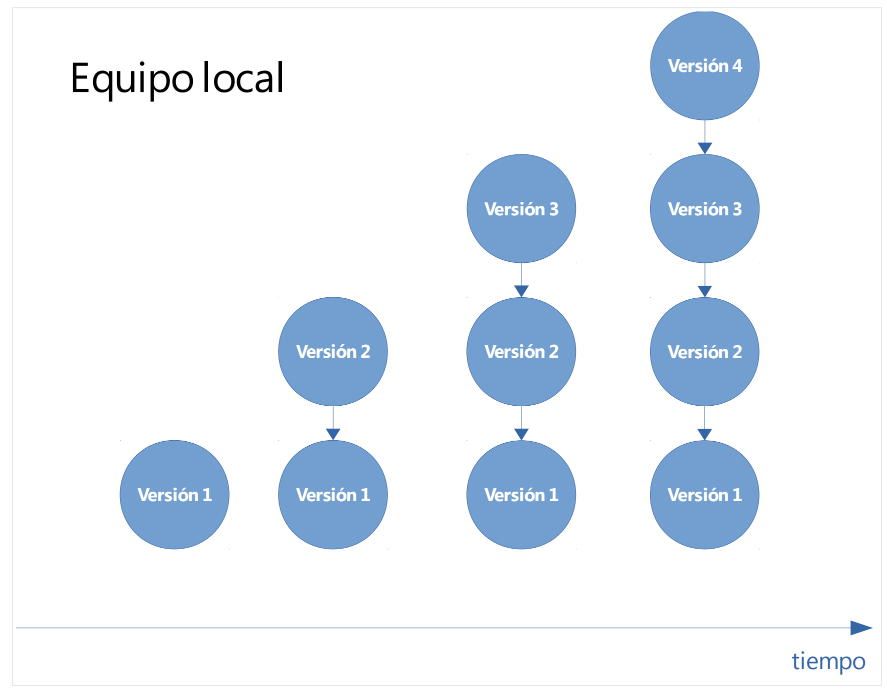


Ilustración 1: Sistema de control de versiones local.

En el año 1982 apareció *Revision Control System* o **RCS**.

Estas herramientas funcionan guardando las diferencias entre las distintas versiones de los archivos. Para poder obtener una versión cualquiera de un archivo lo que hacen es aplicar estas diferencias al archivo base.

En la ilustración 1 podemos ver cómo tenemos 4 versiones de un conjunto de archivos. En cualquier momento podemos regresar a una versión anterior de un archivo o ver las diferencias entre dos versiones distintas.

2.3.2 Control de versiones centralizado

Con la llegada de Internet y la necesidad de colaborar entre distintos programadores aparecen los sistemas control de versiones centralizados, en los que en un único servidor se almacenan las distintas versiones de los archivos. Los clientes que quieren trabajar en un proyecto lo descargan desde el servidor y cuando quieren guardar y compartir los cambios que acaban de introducir los envían al servidor.

Existen muchos sistemas control de versiones centralizados. Entre los más conocidos se encuentran

- *Concurrent Versions System* o **CVS**, que apareció en el año 1986.
- *Microsoft Visual SourceSafe* o **VSS**, que apareció en año 1994.
- *Subversion* o **SVN**, que apareció en el año 2000.

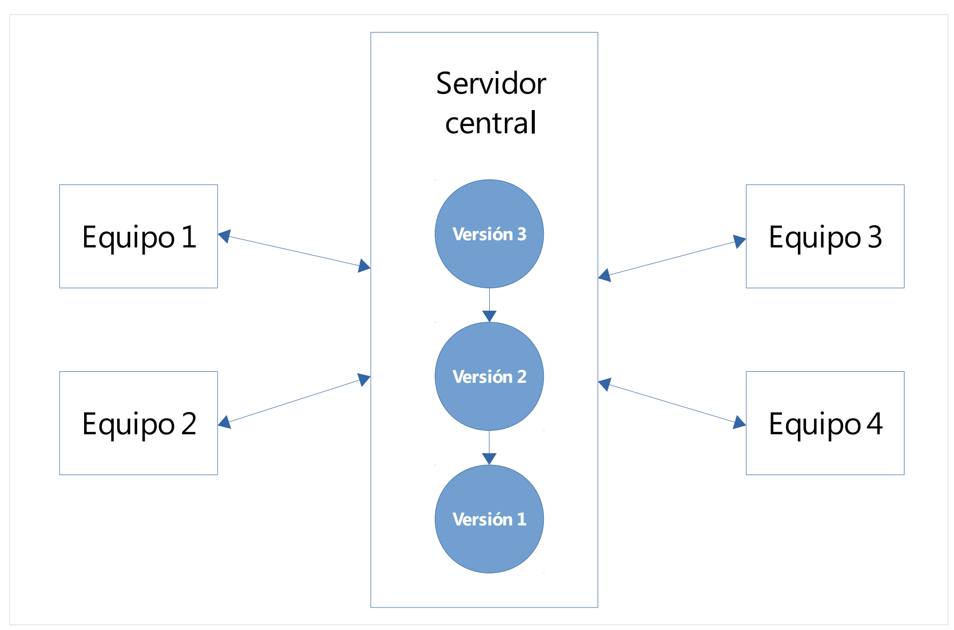


Ilustración 2: Sistema de control de versiones centralizado.

En la ilustración 2 podemos ver como el servidor dispone de 3 versiones de un conjunto de archivos. Los clientes obtienen la última versión de los archivos, trabajan localmente y actualizan los cambios en el servidor.

Esta arquitectura centralizada, aunque presenta muchas ventajas respecto a la local, también tiene sus puntos débiles, ya que toda la información se encuentra en un único punto centralizado; si este punto falla no podemos ni obtener ni compartir la información con el resto de compañeros y el trabajo se complica notablemente; además, si el disco duro del servidor se estropea y no disponemos de copias de seguridad vamos a perder todo el historial del proyecto, ya que lo único que se podrá recuperar es la información puntual de la que disponen los distintos usuarios en su equipo.

2.3.3 Control de versiones distribuido

Para mejorar el funcionamiento de los sistemas control de versiones centralizados aparecen los sistemas control de versiones distribuidos o DVCS (distributed version control system), que toman un enfoque similar al peer-to-peer (P2P), en contraposición con la arquitectura cliente-servidor de los anteriores sistemas.

En lugar de disponer de un único repositorio centralizado al que se sincronizan los clientes, cada equipo posee una copia completa de todo el repositorio y no solo la última instantánea del proyecto.

Si hay algún problema con algún servidor central, establecido por convenio, éste se puede regenerar a partir de cualquier otro cliente.

En la ilustración 3 podemos ver como tanto el servidor, establecido por convenio, ya que no deja de ser un equipo más con algunas características concretas, como el resto de equipos, disponen de un conjunto de versiones del proyecto, de tal forma que pueden trabajar en local sin tener que conectarse a otros equipos, excepto para compartir el trabajo y para obtener el trabajo de sus compañeros. Esto es muy importante, porque que nos permite trabajar sin conexión utilizando toda la potencia de Git, al tener en el equipo local todas las versiones del proyecto. Si, por ejemplo, estamos trabajando en un lugar sin conexión, podremos realizar el trabajo de forma habitual y solo necesitaremos conexión cuando queramos compartir el trabajo con el resto de desarrolladores del proyecto.

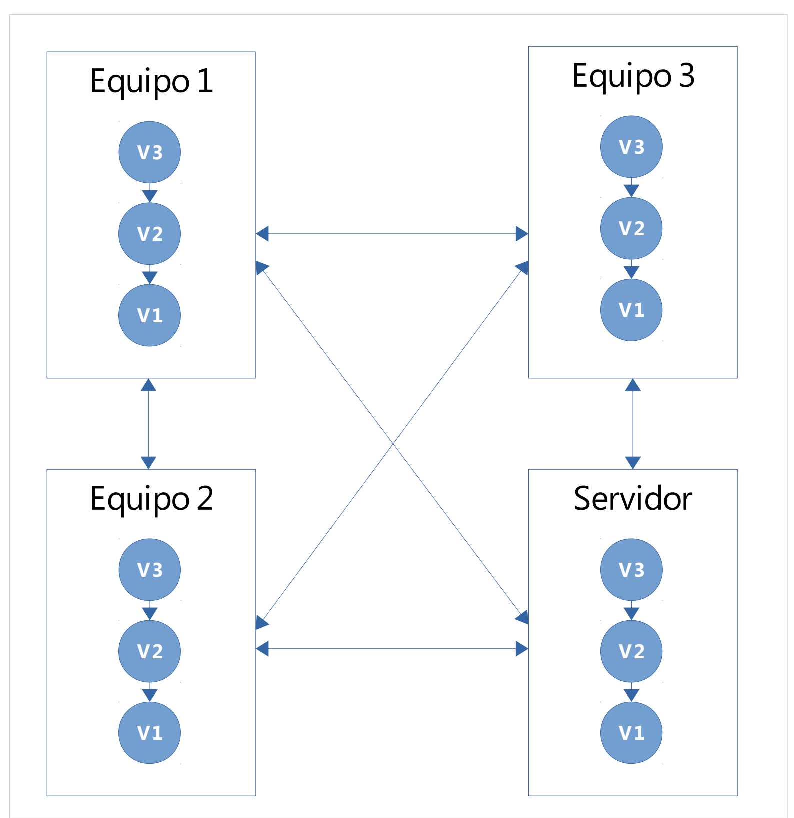


Ilustración 3: Sistema de control de versiones distribuido.

Existen muchos sistemas de control de versiones distribuidos. Entre los más conocidos se encuentran

- ***BitKeeper***, que apareció en el año 1998.
- ***Bazaar***, que apareció en el año 2005.
- ***Mercurial***, que apareció en el año 2005.
- Y el propio ***Git***, que apareció en el año 2005.

2.4 La creación de Git y sus características fundamentales

El kernel del sistema operativo Linux es uno de los proyectos más importantes llevados a cabo de forma colaborativa a través de Internet.

El proyecto lo inició Linus Torvalds en el año 1991.

En el año 2002 empezó a usar BitKeeper, un sistema de control de versiones distribuido con licencia propietaria, que proporcionaba licencias gratuitas a proyectos de software libre.

En el año 2005 la empresa dejó de proporcionar una licencia gratuita, con lo que Linus Torvalds tuvo que buscar alternativas.

Linus buscó un sistema distribuido que pudiera usar del mismo modo que BitKeeper, pero ninguno de los sistemas disponibles con licencia libre cumplía sus necesidades, sobre todo en sus requisitos de rendimiento, por lo que decidió desarrollar uno que le permitiera gestionar el desarrollo del kernel.

Las características fundamentales que implementa Git, fruto de la experiencia de Linus como líder del desarrollo de un gran proyecto distribuido como es el Kernel y de su amplio conocimiento del rendimiento de los sistemas de archivos, son las siguientes:

Rápido y escalable. Git fue desarrollado teniendo como objetivo la gestión del Kernel de Linux, por lo que es un sistema de control de versiones rápido y escalable, adaptándose tanto a proyectos individuales como a grandes proyectos en los que colaboran cientos de desarrolladores, como es el caso del kernel de Linux.

Copia completa. Al igual que otros sistemas de control de versiones distribuidos, Git almacena una copia completa con todo el historial de desarrollo en cada usuario local. Si en algún momento el servidor deja de estar operativo, este puede ser recuperado a partir de cualquier copia local en un equipo de desarrollo que esté actualizado.

Además, en cualquier momento podemos acceder a un estado previo (un commit) o ver las diferencias entre dos commits sin tener que acceder a un equipo externo.

Desarrollo distribuido. Disponer de una copia completa del repositorio facilita enormemente el desarrollo simultáneo e independiente en repositorios privados, evitando la necesidad de sincronización continua con un repositorio central.

Incluso permite trabajar sin conexión a Internet durante periodos prolongados. Una vez que vuelves a tener conexión los cambios son propagados entre los distintos repositorios a través de una red local o de Internet.

Trabajo local. La mayor parte del trabajo que llevamos a cabo con Git lo podemos realizar en un equipo local sin conexión a otras redes. Si queremos ver los cambios introducidos en un archivo en los últimos 15 días o el historial de commits, no tendremos que conectarnos a un equipo remoto, ya que todas estas operaciones se ejecutan localmente.

Solo tendremos que conectarnos a otros equipos cuando queramos colaborar en un proyecto con otros usuarios, compartiendo u obteniendo los cambios introducidos en el proyecto desde la última conexión.

Ramas. Git dispone de un sistema que permite crear ramas y fusionarlas de una forma sencilla y poco costosa, lo que permite un desarrollo no lineal, facilitando la realización de pruebas, parches, resolución de errores,... en distintas ramas.

Instantáneas. La mayoría de sistemas de control de versiones almacena internamente la información como una lista de cambios de los archivos; es decir, guardan los archivos y las modificaciones hechas a lo largo del tiempo.

Git no gestiona sus datos internamente de esta forma, sino que los guarda como un conjunto de instantáneas o fotografías del sistema de archivos en cada commit. Cada vez que un usuario guarda el proyecto en el repositorio Git saca una especie de fotografía del estado puntual del proyecto.

Múltiples protocolos. Git soporta varios protocolos existentes y de uso muy amplio, como son HTTP, HTTPS y SSH. Además incorpora un protocolo propio.

Robustez. Para evitar la corrupción de archivos y los cambios no deseados Git utiliza internamente una función criptográfica de tipo hash denominada SHA-1 (Secure Hash Algorithm), de tal forma que cualquier mínimo cambio será detectado por Git. Si cambiamos un archivo o si este se corrompe Git lo sabrá.

El resultado de un hash SHA-1 es una cadena de 40 caracteres hexadecimales: números entre el 0 y el 9 y letras entre la a y la f. A lo largo de curso veremos los hash SHA-1 por todos lados.

Una de las ventajas que añade los resúmenes o hash son que las búsquedas o comparaciones, en las que no tiene que estar consultando el contenido de cada archivo o diferencia, sino que en muchos casos es suficiente con hacerlo con los SHA-1, con lo que el incremento en el rendimiento de muchas operaciones respecto a otros sistemas de control de versiones es muy grande.

Libre. Siguiendo la filosofía de la mayoría de proyectos colaborativos, Git es libre. Está licenciado

con la licencia GNU GPL 2.

Gratuito. Además de libre es gratuito. Podemos descargar su código fuente, los instaladores y los ejecutables para la mayoría de los sistemas operativos de forma gratuita.

3 Instalación

A continuación vamos a ver cómo podemos instalar Git en los tres principales sistemas operativos existentes: Windows, Linux (Debian y Fedora) y Mac.

3.1 Instalación en Windows

Veamos cómo se instala Git en Windows 7.

Lo primero que vamos a hacer es descargar el instalador del paquete que vamos a utilizar, denominado Git for Windows.

Para ello accedemos a la web oficial del proyecto http://git-scm.com/downloads y hacemos clic en el icono de Windows.



Una vez que ha finalizado la descarga vamos hasta la carpeta donde lo hemos descargado y ejecutamos el instalador haciendo doble clic sobre el archivo.

Aceptamos la licencia GNU GPL 2 y le indicamos la ruta de instalación.

En este caso vamos a instalar Git en "***C:\Program Files (x86)\Git***".

En la ventana de selección de componentes dejamos todo como está:

- No queremos iconos adicionales.
- Queremos integración con el Explorador de Windows.
- Queremos asociar los archivos con extensión .git con el editor de textos que tengamos por defecto.
- Queremos asociar los archivos .sh para ser ejecutados por el intérprete de comandos Bash.
- Y marcamos que se use una fuente TrueType para todas las consolas de Windows.

Luego dejamos el texto "Git" para el menú de Inicio.

En la ventana "PATH environment" seleccionamos la segunda opción, de tal forma que añade Git a la variable de entorno Path de Windows.

En la ventana donde elegimos el ejecutable SSH (solo aparece si tenemos instalado en nuestro

equipo algún programa de conexión SSH, como, por ejemplo, Putty), seleccionamos la primera opción "Use OpenSSH".

En la ventana de conversión de final de línea dejamos seleccionada la primera opción, de tal forma que Git convertirá de forma adecuada los finales de línea en los archivos de texto entre el formato Windows y el formato Unix.

Se inicia la instalación.

Deseleccionamos el cuadro "View ReleaseNotes.rtf", hacemos clic en el botón "Finish" y ya hemos terminado la instalación de Git en Windows.

Para ejecutar Git vamos a Inicio, Todos los programas, Git y seleccionamos Git Bash.

Acabamos de arrancar la versión de Windows del intérprete de comandos Bash. Podemos ejecutar comandos como

```
pwd
ls -la
```

Escribiendo

```
git --version
```

nos indicará la versión del Git que acabamos de instalar.

Si vamos a Inicio, Todos los programas, Git y seleccionamos Git GUI accedemos a la interfaz gráfica que instala Git por defecto, desde la que también podremos trabajar.

Si accedemos mediante el Explorador de Windows a una carpeta cualquiera y hacemos clic con el botón derecho sobre el fondo podemos ver 3 elementos:

- ***Git Init Here***, que inicializaría un repositorio Git en este directorio.
- ***Git Gui***, que ejecutaría la interfaz gráfica tomando este directorio como base.
- ***Git Bash***, que ejecutaría el intérprete de comandos, tomando como base el directorio actual, tal y como acabamos de ver.

3.2 Instalación en Linux

A continuación vamos a ver cómo se instala Git en Linux, tanto en una versión Debian como en una Fedora, ya que usan diferentes tipos de paquetes y son las dos variantes más utilizadas.

3.2.1 Ubuntu

Empezaremos por una máquina Ubuntu. De una forma casi idéntica se puede realizar la instalación en distribuciones que usen el formato de paquete .deb: Debian, Linux Mint, ...

Accedemos a una consola y verificamos que Git no se encuentre instalado ejecutando

```
git
```

En caso de que no se encuentre el programa pasamos a instalarlo, ejecutando

```
sudo apt-get update
```

y posteriormente

```
sudo apt-get upgrade
```

para actualizar el listado de paquetes disponibles e instalar las últimas actualizaciones. De esta forma tenemos el sistema actualizado. Estos dos pasos son opcionales, aunque recomendables.

Ejecutamos

```
sudo apt-get install git
```

para instalar Git en nuestra máquina.

Tras finalizar la instalación ejecutamos

```
git --version
```

```
    git version 2.1.4
```

para comprobar que se ha instalado correctamente.

Si queremos instalar todos los subpaquetes complementarios para Git, ejecutamos

```
sudo apt-get install git-all
```

que instalará en nuestro sistema paquetes para interoperar con otros sistemas de control de versiones como Arch, CVS, Subversion o Bazaar, con wikis como MediaWiki o interfaces gráficas, entre otras funcionalidades.

Si queremos tener instalada la última versión estable de Git lo tendremos que hacer mediante un PPA, un Personal Package Archive.

En este caso vamos a añadir uno que se encarga de mantener la última versión estable de Git. Ejecutamos:

```
sudo add-apt-repository ppa:git-core/ppa
```

Actualizamos el repositorio local:

```
sudo apt-get update
```

E instalamos el paquete Git:

```
sudo apt-get install git
```

O lo actualizamos si ya lo tenemos instalado:

```
sudo apt-get upgrade
```

3.2.2 CentOS

Ahora vamos a ver cómo se instala Git en una distribución CentOS. De una forma casi idéntica se puede realizar la instalación en distribuciones basadas en Red Hat, como Red Hat Enterprise Linux,

Fedora, ...

Accedemos a una consola y verificamos que Git no se encuentre instalado ejecutando

```
git
```

Luego ejecutamos

```
sudo yum upgrade
```

para instalar las últimas actualizaciones. De esta forma tenemos el sistema actualizado. Este paso es opcional, aunque recomendable.

Ejecutamos

```
sudo yum install git
```

para instalar Git en nuestra máquina.

Tras finalizar la instalación ejecutamos

```
git --version
```

para comprobar que se ha instalado correctamente.

Si queremos instalar todos los subpaquetes complementarios para Git, ejecutamos

```
sudo yum install git-all
```

que instalará en nuestro sistema paquetes para interoperar con otros sistemas de control de versiones como Arch, CVS, Subversion o Bazaar, con wikis como MediaWiki o interfaces gráficas, entre otras funcionalidades.

3.3 Instalación en OS X

Para instalar Git en OS X podemos hacerlo de dos formas:

1) Mediante el instalador **git-osx-installer**, que está disponible para su descarga en http://git-scm.com/download/mac

Una vez descargado tenemos que realizar su instalación como cualquier otro paquete.

2) Mediante MacPorts.

Tras instalar MacPorts, disponible en http://www.macports.org, tenemos que ejecutar en una consola:

```
sudo port install git-core +svn +doc +bash_completion +gitweb
```

4 Primeros pasos

4.1 Obtener ayuda en Git

Para comenzar con Git vamos a ver cómo podemos utilizar su ayuda.

Si ejecutamos desde la consola

```
git
```

sin ningún parámetro aparece una pequeña ayuda con todos los parámetros que puede recibir y con los comandos más utilizados, seguido de una pequeña explicación.

```
usage: git [--version] [--exec-path[=<path>]] [--html-path] [--man-path] [--
info-path]
           [-p|--paginate|--no-pager] [--no-replace-objects] [--bare]
           [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
           [-c name=value] [--help]
           <command> [<args>]
The most commonly used git commands are:
   add        Add file contents to the index
   bisect     Find by binary search the change that introduced a bug
   branch     List, create, or delete branches
   checkout   Checkout a branch or paths to the working tree
   clone      Clone a repository into a new directory
   commit     Record changes to the repository
   diff       Show changes between commits, commit and working tree, etc
   fetch      Download objects and refs from another repository
   grep       Print lines matching a pattern
   init       Create an empty git repository or reinitialize an existing one
   log        Show commit logs
   merge      Join two or more development histories together
   mv         Move or rename a file, a directory, or a symlink
   pull       Fetch from and merge with another repository or a local branch
   push       Update remote refs along with associated objects
   rebase     Forward-port local commits to the updated upstream head
   reset      Reset current HEAD to the specified state
   rm         Remove files from the working tree and from the index
   show       Show various types of objects
   status     Show the working tree status
   tag        Create, list, delete or verify a tag object signed with GPG
See 'git help <command>' for more information on a specific command.
```

Por ejemplo, podemos ver lo que hace el comando init, que permite crear un repositorio git vacío o

reinicia uno existente.

En la parte inferior vemos que podemos ejecutar “git help” seguido del comando o concepto que nos interesa para obtener más información sobre ese comando o concepto concreto.

Por ejemplo, si queremos obtener información sobre el comando init, con el que podemos crear un repositorio Git vacío, ejecutamos el comando

```
git help init
```

Este comando nos muestra el manual de ayuda de Git. Podemos ver una breve descripción en la parte superior, en el apartado del nombre, donde nos indica que este comando crea un repositorio Git vacío o reinicia uno existente.

A continuación muestra los posibles parámetros que puede recibir el comando.

En la parte inferior muestra una descripción completa de las operaciones que realiza el comando.

Para finalizar podemos ver los distintos parámetros que puede recibir este comando, como por ejemplo “*--quiet*”, que hace que el comando “git init” solo muestre mensajes de error y warnings.

Para avanzar una ventana en la ayuda utilizamos la *barra espaciadora* o la letra *f*, inicial de la palabra inglesa *Forward*.

Para retroceder una ventana en la ayuda utilizamos la letra *b*, inicial de la palabra inglesa *Backward*.

Si queremos salir de la ayuda simplemente tenemos que pulsar la letra *q*, inicial de la palabra inglesa *Quit*.

Otra forma que tenemos para ejecutar la ayuda es utilizando el parámetro “*--help*” después del comando. En el ejemplo que estamos siguiendo del comando init ejecutaríamos

```
git init --help
```

y podemos ver que el resultado es el mismo que el del comando anterior.

Para aquellos usuarios que estáis acostumbrados a trabajar en entornos Unix, otra forma de ejecutar la ayuda es utilizar el comando *man* seguido de *git*, un guion o un espacio y el comando.

En este caso, para el ejemplo visto ejecutaríamos

```
man git-init
```

o

```
man git init
```

y podemos ver que el resultado es el mismo que el del comando inicial.

Este último comando (*man*) no funciona en un entorno Windows.

Si no eres capaz de encontrar la ayuda a través de las distintas páginas de ayuda siempre puedes tratar de obtener ayuda utilizando el IRC en Freenode, concretamente en los canales #git y #github.

En estos canales suele haber decenas de personas dispuestas a echar una mano con las dudas que tengas; eso sí, en inglés.

Si queremos obtener una lista completa con todos los subcomandos disponibles tenemos que ejecutar

```
git help -a
```

y obtendremos la lista completa.

Si queremos obtener una lista con las guías de conceptos más habituales tenemos que ejecutar

```
git help -g
```

Una de las guías de conceptos que aparece es “glossary”. Para acceder a ella ejecutamos

```
git help glossary
```

y tendemos acceso a una lista de los términos más habituales de Git.

En Windows, por defecto, la ayuda se abre en el navegador, en vez de mostrarse en la consola, como lo hace en Linux.

Por lo tanto, las formas que tenemos de obtener ayuda mediante la línea de comandos son

```
git help [comand]
git [comando] --help
man git-[comando]
man git [comando]
```

4.2 Configuración inicial de Git

Tras instalar Git en nuestro sistema, antes de empezar a trabajar con él tenemos que realizar una pequeña configuración inicial.

Solo la tenemos que realizar una vez y los valores que vamos a actualizar los podemos cambiar a posteriori.

El comando que utilizaremos para realizar la configuración de git es

```
git config
```

que permite establecer valores en determinadas variables de configuración que controlan Git, así como acceder a sus valores.

Estas variables que almacenamos de forma permanente pueden ser guardadas en tres niveles:

A **nivel de sistema**, de tal forma que esos valores se utilizarán para cada usuario y para cada repositorio en el sistema.

Tenemos que utilizar el parámetro “*--system*” para que los parámetros usados se almacenen a nivel de sistema.

Esta información se almacena en el archivo “/etc/gitconfig” si estamos trabajando en sistemas Unix.

Si estamos en un sistema Windows este archivo estará también en “/etc/gitconfig”, pero de forma relativa a donde hemos instalado Git.

Por ejemplo, si lo hemos instalado en “C:\Program Files (x86)\Git\”, la información se almacena en el archivo “C:\Program Files (x86)\Git\etc\gitconfig”.

A **nivel de usuario**, de tal forma que esos valores se utilizarán para todos los repositorios en los

que trabaje ese usuario concreto.

Tenemos que utilizar el parámetro "--global" para que los parámetros usados se almacenen a nivel de usuario.

Esta información se almacena en el archivo ".gitconfig"en el directorio raíz del usuario.

Por ejemplo, si estamos en un sistema Unix y nuestro usuario es "fonteLearn", el archivo estará en la ruta "/home/fonteLearn/.gitconfig".

Tenemos que ejecutar el comando "ls -a" para visualizar ese archivo, ya que empieza con un punto.

Si estamos en un sistema Windows este archivo estará en "C:\Users\Mi usuario", donde "Mi usuario" se corresponde con el usuario con el que hemos iniciado la sesión.

Ejecutando

```
dir /p
```

dentro del directorio del usuario podemos ver ese archivo.

A **nivel de proyecto**, de tal forma que esos valores solo se usarán para ese proyecto concreto.

Es el nivel predeterminado, con lo que no tenemos que utilizar ningún parámetro.

Esta información se guarda en el archivo "*config*" dentro del directorio ".git" que se crea en la raíz del repositorio, como veremos más adelante.

En caso de conflicto en un mismo parámetro, la configuración más prioritaria es a nivel de proyecto y la menos prioritaria a nivel de sistema.

4.2.1 Identidad

Lo primero que tenemos que configurar es el **usuario** y el **correo electrónico** del usuario, de tal forma que esta información pueda ser utilizada en cada commit, identificando quién lo ha realizado, para poder tener una trababilidad total de los cambios introducidos.

El primer parámetro que configuro es el nombre del usuario, ejecutando

```
git config --global user.name "Fonte Learn"
```

A continuación configuro el correo electrónico, ejecutando

```
git config --global user.email test@fontelearn.com
```

Al utilizar el parámetro "--global" esta información queda guardada en el archivo de configuración del usuario, de tal forma que todo lo que haga en el sistema utilizará esta información.

Si en algún repositorio concreto queremos cambiar alguno de estos parámetros, podemos volver a ejecutar el comando, pero sin el parámetro "--global", afectando este cambio solo al repositorio.

4.2.2 Editor

El siguiente parámetro que vamos a configurar es el editor que usaremos cuando Git necesite que escribamos un mensaje. Git usa el editor por defecto del sistema, pero si lo queremos cambiar ejecutamos

```
git config --global core.editor emacs
```

para usar *emacs*

O

```
git config --global core.editor vim
```

para utilizar *vim*.

En Windows podemos utilizar, por ejemplo, Notepad, que viene instalado con el sistema operativo, usando el comando

```
git config --global core.editor notepad.exe
```

4.2.3 Otros elementos

También podemos configurar la herramienta que usaremos para mostrar las diferencias entre dos archivos y como herramienta de resolución de conflictos. Si, por ejemplo, queremos utilizar la herramienta "vimdiff", ejecutaremos el comando

```
git config --global diff.tool vimdiff
git config --global merge.tool vimdiff
```

Si queremos configurar Git para que use colores en la salida y de esta forma mostrar una información mucho más significativa y usable, utilizaremos el comando

```
git config --global color.ui true
```

Cuando trabajamos en entornos mixtos, con equipos de desarrollo Windows y Unix (Mac y Linux) podemos tener problemas con los archivos porque los finales de línea son tratados de forma distinta en Windows, donde se añade un final de línea y un retorno de carro y en Unix, donde solo se añade un final de línea.

Para evitar estos problemas es aconsejable que Git se encargue de hacer las conversiones necesarias, con lo que si estamos en Windows ejecutaremos el comando

```
git config --global core.autocrlf true
```

de tal forma que Git realiza la conversión automática de los finales de línea.

En los equipos Unix no tenemos ese problema, pero si accidentalmente recibimos un archivo con formato Windows querremos que se lleve a cabo de forma automática la conversión, con lo que es aconsejable ejecutar el comando

```
git config --global core.autocrlf input
```

con lo que solo se realizará la conversión en los archivos que recibimos desde otro equipo.

Para consultar los valores de cualquiera de estos parámetros usaremos el comando "git config" seguido del parámetro que queremos consultar.

Por ejemplo, para consultar el valor de estos parámetros ejecutamos

```
git config user.name
```

para poder visualizar el nombre

```
Fonte Learn
```

y

```
git config user.email
```

para mostrar el correo electrónico

```
test@fontelearn.com
```

Si queremos ver todos los parámetros que tenemos configurados ejecutamos

```
git config --list
```

Para finalizar, si queremos visualizar el contenido del archivo donde se guarda la configuración de usuario, ejecutamos

```
cd ~
```

para situarnos en el directorio raíz del usuario

y luego ejecutamos

```
cat .gitconfig
```

4.3 Conceptos básicos

Antes de empezar a trabajar con Git vamos a ver una serie de conceptos básicos, que nos permitirán establecer los fundamentos para poder ir avanzando sin complicaciones. Los elementos que vamos a ver en este apartado son:

- Qué es un repositorio.
- Qué es un commit.
- Qué zonas utilizamos en Git.
- En qué estado puede estar un archivo o un directorio.
- Flujo de trabajo habitual.
- Qué es un SHA-1.
- Qué es el HEAD.
- Qué es una rama.

4.3.1 Repositorio

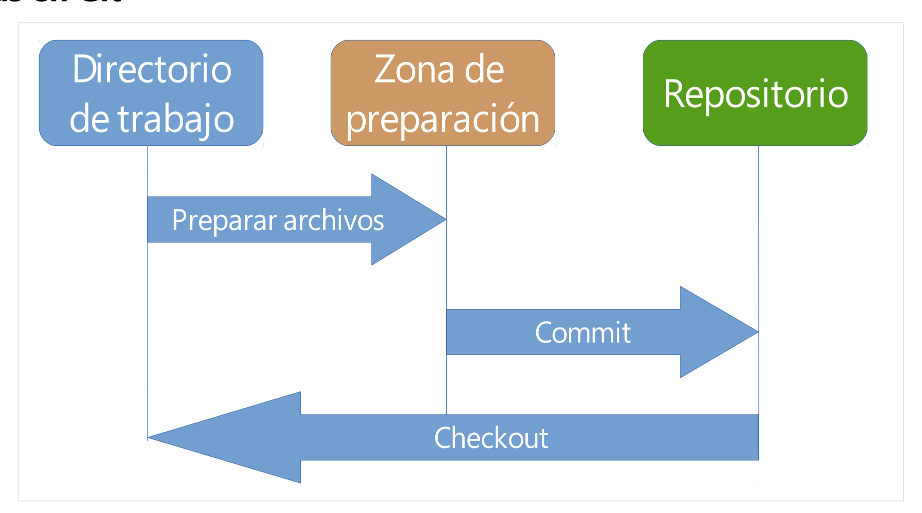
El repositorio es una especie de contenedor o base de datos que almacena, entre otros elementos, un histórico de todos los cambios que se han producido en los archivos del proyecto y que se han depositado en él mediante un commit o confirmación.

4.3.2 Commit

Cuando hacemos un commit o confirmación estamos grabando en el repositorio una instantánea del estado de proyecto, junto con más información, como el autor de los cambios, cuándo se han realizado y una explicación de los cambios que se están introduciendo.

Al almacenar instantáneas de los archivos que forman parte del proyecto podemos saber cuáles han sido los cambios realizados en un archivo entre dos o más commits o quién los ha introducido, entre muchas otras operaciones.

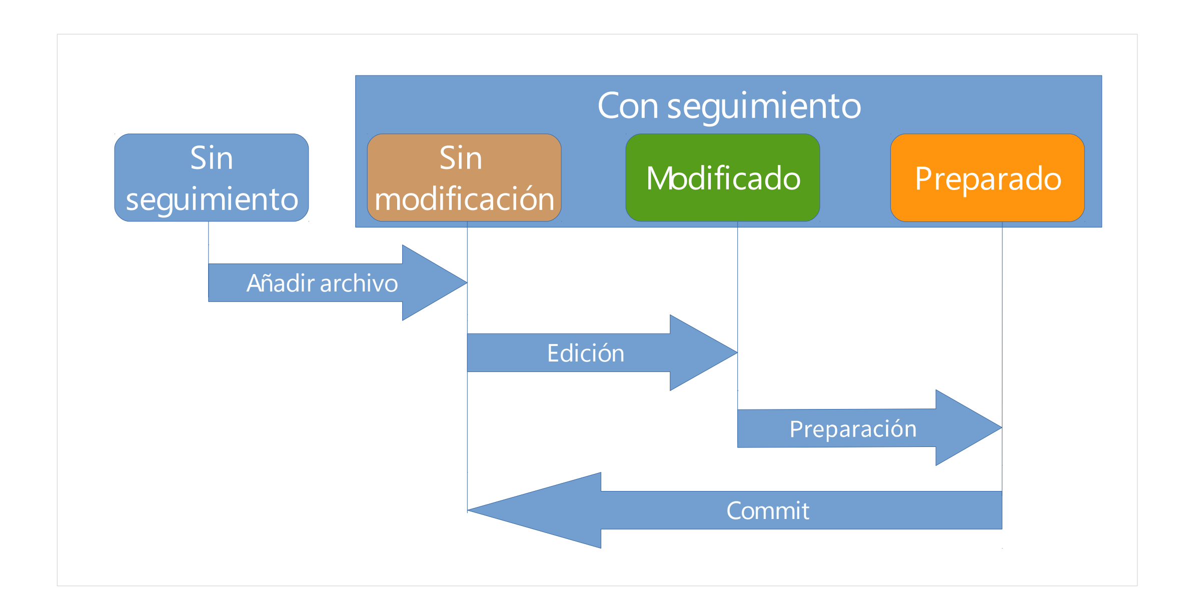
4.3.3 Zonas en Git



Cuando estamos trabajando en un proyecto de forma continua con Git, tenemos tres zonas **locales** en las que puede estar una versión concreta de un archivo:

- **Directorio de trabajo**, también conocido como "*working directory*", es la zona del repositorio que podemos ver cuando ejecutamos "ls -la" en Unix o "dir /p" en Windows y sobre la que podemos trabajar creando, editando, moviendo o borrando los archivos que se encuentran en ese directorio.
- **Zona de preparación**, también conocida con "*staging area*", "*index*" o "*índice*", es una zona intermedia, no accesible directamente desde la interfaz de usuario, pero sí usando Git, donde se encuentran las instantáneas de los archivos que se guardarán en el repositorio en el próximo commit.
- **Repositorio**, también conocido como "repository", que, como hemos visto en el apartado 4.3.1, es una especie de contenedor o base de datos que almacena, entre otros elementos, un histórico de todos los cambios que se han producido en los archivos del proyecto y que se han depositado en él mediante un commit.

4.3.4 Estados de un archivo



Cada uno de los archivos presentes en el directorio de trabajo o en un subdirectorio puede estar **sin seguimiento** o **bajo seguimiento** por parte de Git.

Dentro de los archivos que tenemos bajo seguimiento dentro de un proyecto, estos están en uno de los tres siguientes estados:

- **Sin modificación**, en el que el contenido del archivo que está en la zona de trabajo, en la zona de preparación y en el repositorio es el mismo.
- **Modificado**, en el que el contenido del archivo que está en la zona de trabajo difiere del que está en la zona de preparación y en el repositorio.
- **Preparado**, en el que el contenido del archivo que está en la zona de trabajo coincide con la zona de preparación, pero difiere del repositorio.

4.3.5 Flujo de trabajo

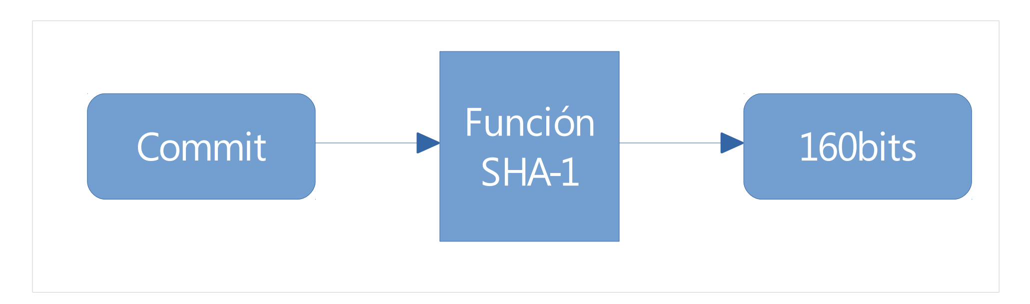
El flujo de trabajo habitual será el siguiente, una vez inicializado el repositorio (apartado 4.4):

1. Creamos un archivo en el directorio del proyecto o en un subdirectorio. En este momento su estado es "*sin seguimiento*".
2. Añadimos el archivo al proyecto, ya que inicialmente éste no se encuentra bajo seguimiento por Git en el proyecto. En este momento su estado es *bajo seguimiento*, pero *modificado*, ya que el contenido de ese archivo en el directorio de trabajo difiere del que está en la zona de preparación y en el repositorio, ya que aún no se encuentra en ninguna de esas dos zonas.
3. Editamos el archivo. Editamos el archivo, con un editor de texto, un IDE, un editor gráfico, ... dependiendo del tipo de archivo con el que estemos trabajando. El archivo sigue estando *modificado*.
4. Preparamos el archivo, añadiendo una *instantánea* del archivo a la zona de preparación. En este momento el archivo está *preparado* para ser incorporado al repositorio en el siguiente commit. El contenido del archivo que está en la zona de trabajo coincide con la zona de preparación, pero difiere del repositorio.

5. Llevamos a cabo el commit, pasando el archivo a estar *sin modificación*, ya que ahora mismo el contenido del archivo que está en la zona de trabajo, en la zona de preparación y en el repositorio es el mismo.
6. A partir de ahora volvemos al punto 3, editando el archivo, excepto que queramos borrarlo, moverlo o excluirlo del seguimiento por parte de Git.

4.3.6 SHA-1

Cuando usamos Git vemos por todos lados una cadena de números y letras que en principio parece que no tienen sentido. Este conjunto de caracteres alfanuméricos no es nada más que una función criptográfica de tipo hash denominada SHA-1 (Secure Hash Algorithm), que a partir de un número cualquiera de bytes (por ejemplo, de un conjunto de objetos que forman un commit) tiene como resultado 20 bytes (160 bits).

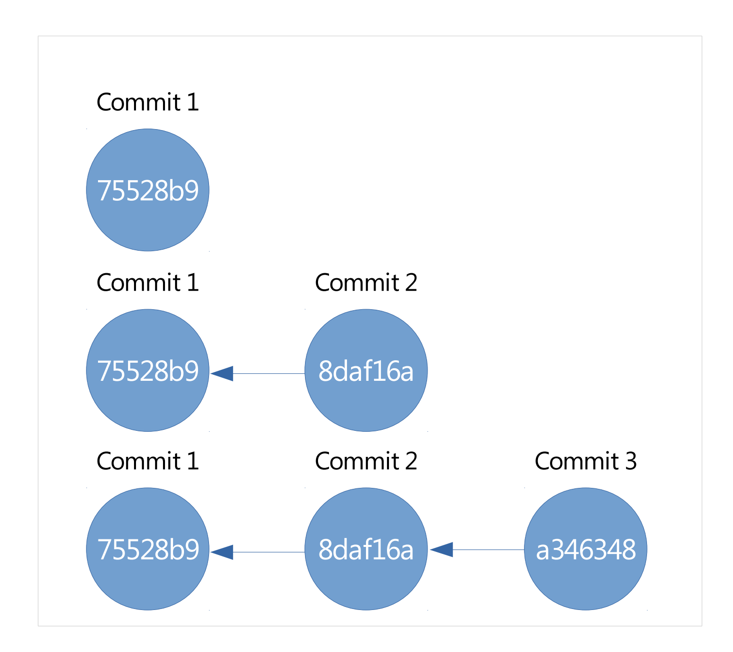


Lo que caracteriza a este tipo de funciones es que cualquier cambio mínimo (desde un simple carácter) da lugar a un SHA-1 totalmente distinto, por lo que Git detectará el cambio. Si cambiamos un archivo o si este se corrompe Git lo sabrá.

El resultado de un hash SHA-1 es una cadena de 40 caracteres hexadecimales (20 bytes): números entre el 0 y el 9 y letras entre la a y la f. Veremos SHA-1 por todos lados, como por ejemplo:

- 8daf16ab0a6b79ef1326f1dcee7a677a206b2128, que es, por ejemplo, el resultado de aplicar la función SHA-1 a un commit, que identificará unívocamente a este commit. Es casi imposible que tengamos otro commit u otro objeto distinto a este que tengan el mismo SHA-1.
- 8daf16a, que es la versión corta del anterior SHA-1 (los primeros 7 caracteres) y que, en la mayoría de proyectos, nos valdrá para identificar unívocamente a un objeto, como puede ser un commit. En otros de tamaño medio o grande puede que tengamos que usar más caracteres.

4.3.7 HEAD



El flujo de trabajo, tal y como hemos visto en el apartado 4.3.5, consiste en ir realizando cambios en los archivos que están bajo seguimiento e ir almacenando sus instantáneas en el repositorio mediante commits, que son identificados de forma única mediante un SHA-1.

La primera vez que realizamos un commit, referenciado por el SHA-1 75528b9, éste no tiene, como es obvio, ningún commit previo.

El segundo commit, referenciado por el SHA-1 8daf16a, tiene como padre el commit 75528b9, apuntando internamente a este, para saber qué commit es su antecesor. Cualquier cambio mínimo en el primer commit tendrá como consecuencia que su SHA-1 varíe, y en consecuencia el SHA-1 del segundo commit.

El tercer commit, referenciado por el SHA-1 a346348, es el último commit del proyecto. El HEAD simplemente es un puntero o referencia que apunta a este último commit. En el apartado 5.2 veremos donde se almacena.

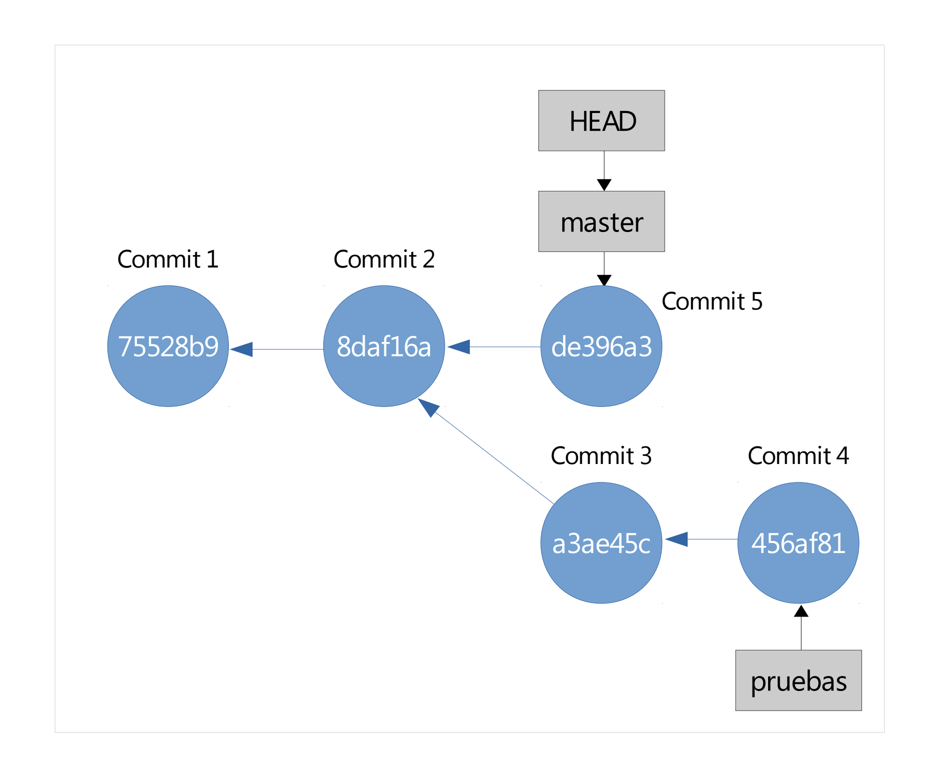
4.3.8 Rama

Una rama (branch) es una bifurcación o camino alternativo que seguimos en un determinado momento para, por ejemplo, arreglar un bug o probar una idea sobre la que no estamos seguros de que vaya a ser viable.

Git, por defecto, crea una rama denominada master; pero podemos crear más ramas y cambiarnos entre ellas, de tal forma que el contenido del directorio de trabajo irá cambiando.

En la ilustración 9 podemos ver un ejemplo en el que:

- Hemos creado dos commits (75528b9 y 8daf16a) en la rama principal, la *master*.
- A continuación hemos creado una nueva rama, *pruebas*, en la que hemos introducido dos nuevos commits (a3ae45c y 456af81).
- Luego nos hemos cambiado a la rama *master*, y hemos introducido un nuevo commit (de396a3).
- Podemos ver que HEAD apunta al último commit de la rama *master*, que es la rama activa.


HEAD
master
Commit 1
Commit 2
Commit 5
75528b9
8daf16a
de396a3
Commit 3
Commit 4
a3ae45c
456af81
pruebas

4.4 Inicializando un repositorio

Para poder trabajar con un repositorio Git antes de nada tenemos que inicializarlo; es decir, preparar el espacio de trabajo para que Git trabaje adecuadamente. Esta tarea podemos llevarla a cabo:

- Clonando un repositorio existente.
- A partir de un directorio existente.
- A partir de un directorio nuevo.

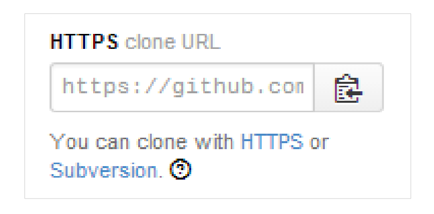
4.4.1 Clonando un repositorio existente

Clonar un repositorio no es nada más que traer a nuestro equipo el espacio de trabajo remoto y el repositorio remoto completo, donde podemos ver todos los cambios que se han producido en cada archivo, ya que recibimos todos los commits del proyecto y otra serie de elementos necesarios.

La forma habitual de usar el comando es

```
git clone URL [directorio]
```

Por ejemplo, si queremos traer a nuestro equipo una copia del proyecto jQuery, que mantiene su código fuente en GitHub, accederemos mediante un navegador a esta página y en la parte derecha vemos un cuadro



Este cuadro tiene una URL en el cuadro de texto. Esa es la URL que usaremos para clonar el proyecto. Para ello ejecutamos

```
cd ~/proyectos/
git clone https://github.com/jquery/jquery.git
```

Podemos ver cómo se clona el repositorio en nuestro equipo

```
Cloning into 'jquery'...
remote: Reusing existing pack: 32992, done.
remote: Total 32992 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (32992/32992), 19.39 MiB | 4.38 MiB/s, done.
Resolving deltas: 100% (23383/23383), done.
```

Si ahora vemos el contenido del directorio "~/proyectos/" ejecutando

```
ls
```

Podemos ver que aparece el siguiente directorio

```
jquery
```

Si vemos su contenido mediante el comando

```
ls jquery -la
```

Podemos ver varios elementos interesantes dentro del direcotiro de trabajo, que son:

- *src*, que es el directorio donde se almacena el código fuente del proyecto.
- .git, que es el directorio donde almacena toda la información del repositorio.
- .gitattributes, que es un archivo de configuración que permite definir atributos a determinadas rutas.
- .gitignore, que también es un archivo de configuración, donde indicaremos aquellos archivos y directorios que no queremos que sean seguidos por Git dentro del proyecto.

```
AUTHORS.txt
bower.json
.bowerrc
build
CONTRIBUTING.md
.editorconfig
.git
.gitattributes
.gitignore
Gruntfile.js
.jscs.json
.jshintignore
.jshintrc
.mailmap
MIT-LICENSE.txt
.npmignore
package.json
README.md
src
test
.travis.yml
```

Si ejecutamos el comando "*git log*", que veremos con detalle en su capítulo correspondiente, pero que, a grandes rasgos permite mostrar el histórico de los commits

```
cd jquery
```

```
git log --format='%Cred%h%Creset -%C(yellow)%d%Creset %s
%Cgreen(%cr)' --abbrev-commit --date=relative -10
```

podemos ver los últimos 10 commits del proyecto.

```
99d735a - (HEAD, origin/master, origin/HEAD, master) Build: change
.gitattributes; use system line ends for non-JS files (4 days ago)

4f490e5 - Support: Add Android support tests results (8 days ago)

5265cda - Build: Update release script for new jquery-release API (3 weeks
ago)

2c2c93c - Selector: Use Element.matches in selector-native if available (3
weeks ago)

ad032d3 - Event: Fix isDefaultPrevented for bubbled events in Android 2.3 (3
weeks ago)

890d441 - Effects: Don't overwrite display:none when .hide()ing hidden
elements (3 weeks ago)

5a8f769 - CSS: jQuery#hide should always save display value (3 weeks ago)

85af4e6 - Manipulation: Change support test to be WWA-friendly (3 weeks ago)

541e734 - Attributes: Trim whitespace from option text when returned as a
value (3 weeks ago)

e547a27 - CSS: window.getDefaultComputedStyle may return null (3 weeks ago)
```

También podíamos haber clonado el repositorio a un directorio distinto, ejecutando el comando

```
cd ~/proyectos/

git clone https://github.com/jquery/jquery.git mi_jquery
```

Podemos ver cómo se clona el repositorio en nuestro equipo

```
Cloning into 'mi_jquery'...

remote: Reusing existing pack: 32992, done.

remote: Total 32992 (delta 0), reused 0 (delta 0)

Receiving objects: 100% (32992/32992), 19.39 MiB | 4.73 MiB/s, done.

Resolving deltas: 100% (23383/23383), done.
```

Si mostramos el contenido del directorio ejecutando

```
ls -1
```

Podemos ver que ahora tenemos dos directorios

```
jquery

mi_jquery
```

4.4.2 A partir de un proyecto existente

Tras ver como se clona un proyecto, lo siguiente que vamos a ver es cómo podemos inicializar un

repositorio a partir de un proyecto existente.

Para ello accedemos a nuestro directorio de proyectos y creamos un directorio nuevo.

```
cd ~/proyectos/
mkdir proyecto_existente
cd proyecto_existente
```

Ahora creamos un par de archivos para simular que este directorio tiene contenido antes de que inicialicemos su repositorio:

```
touch archivo_a.txt
touch archivo_b.txt
```

Si ejecutamos

```
ls -1
```

Vemos que por ahora en ese directorio solo tenemos los dos archivos creados mediante el comando touch.

```
archivo_a.txt
archivo_b.txt
```

Para inicializar el repositorio en ese directorio simplemente tenemos que ejecutar el comando

```
git init
```

```
Initialized empty Git repository in
/home/fontelearn/proyectos/proyecto_existente/.git/
```

Si ahora volvemos a visualizar el contenido del directorio

```
ls -a -1
```

Aparece un nuevo directorio, "*.git*"

```
archivo_a.txt
archivo_b.txt
.git
```

donde se almacena toda la información del repositorio. El contenido actual de este directorio es el siguiente

```
.git/
├── branches
```

```
├── config
├── description
├── HEAD
├── hooks
│   ├── applypatch-msg.sample
│   ├── commit-msg.sample
│   ├── post-update.sample
│   ├── pre-applypatch.sample
│   ├── pre-commit.sample
│   ├── prepare-commit-msg.sample
│   ├── pre-rebase.sample
│   └── update.sample
├── info
│   └── exclude
├── objects
│   ├── info
│   └── pack
└── refs
    ├── heads
    └── tags
```

Habitualmente no trabajaremos directamente con estos archivos, sino que los manipularemos mediante comandos de Git, pero siempre es interesante conocer la existencia de este directorio y poder ver dónde almacena Git toda su información.

Si ahora ejecutamos el comando

```
git status
```

que nos muestra el estado de los archivos y directorios que se encuentran en el directorio de trabajo, podemos ver que los dos archivos se encuentran en estado "**sin seguimiento**", uno de los estados que hemos visto en el apartado "*4.3.4 Estados de un archivo*".

```
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       archivo_a.txt
#       archivo_b.txt
```

```
nothing added to commit but untracked files present (use "git add" to track)
```

En el siguiente apartado veremos cómo añadir estos archivos al repositorio.

4.4.3 A partir de un proyecto nuevo

Tras ver como clonar un directorio y como inicializar un proyecto existente, ahora vamos a ver cómo podemos inicializar un proyecto vacío y cómo trabajamos de forma habitual con un repositorio.

Lo primero que hacemos es crear el directorio y acceder a él

```
mkdir ~/proyectos/proyecto_nuevo/
cd ~/proyectos/proyecto_nuevo/
```

Si ejecutamos

```
git status
```

para conocer el estado del repositorio obtendremos un error, ya que aún no hemos inicializado el repositorio

```
fatal: Not a git repository (or any parent up to mount point /home)
Stopping at filesystem boundary (GIT_DISCOVERY_ACROSS_FILESYSTEM not set).
```

Si mostramos el contenido del directorio veremos que está vacío

```
ls -a -1
```

```
.
..
```

Lo que vamos a hacer a continuación es inicializar el repositorio, ejecutando el mismo comando que en el apartado anterior

```
git init
```

```
Initialized empty Git repository in
/home/fontelearn/proyectos/proyecto_nuevo/.git/
```

Si ahora volvemos a visualizar el contenido del directorio

```
ls -la
```

Aparece un nuevo directorio, "*.git*"

```
.
..
.git
```

que es el similar al generado en el apartado anterior, y donde se almacena toda la información del repositorio.

4.5 Añadiendo un primer archivo

Tras clonar o inicializar el repositorio, lo siguiente que vamos a hacer es añadir archivos para que sus cambios sean controlados por el repositorio.

Para ello ejecutamos el comando

```
touch archivo_a.txt
```

Si visualizamos el espacio de trabajo

```
ls -la
```

podemos ver que ahora tenemos este nuevo archivo (omitimos los directorios "." y "..").

```
archivo_a.txt
.git
```

pero si mostramos el estado del repositorio, ejecutando

```
git status
```

```
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    archivo_a.txt

nothing added to commit but untracked files present (use "git add" to track)
```

Podemos ver que hay un archivo "*archivo_a.txt*" que no está siendo seguido por Git.

Para que pase a estar seguido Git y, a la vez, que pase a la zona de preparación, ejecutamos el comando "*git add*" seguido del conjunto de archivos sobre los que queremos ejecutar la operación

```
git add archivo_a.txt
```

También podríamos haber ejecutado el comando

```
git add .
```

De tal forma que todos los archivos que no están siendo seguidos pasarían a estarlo.

Si ahora volvemos a ver el estado del repositorio, ejecutando

```
git status
```

```
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

   new file:   archivo_a.txt
```

Lo que nos está indicando es que tenemos un archivo en estado "**preparado**", uno de los estados que hemos visto en el apartado "*4.3.4 Estados de un archivo*". El archivo está listo para enviar al repositorio mediante un commit, de tal forma que en el repositorio se almacenará esta instantánea del archivo, que podremos recuperar en cualquier momento o usar para realizar comparaciones, ver la evolución del proyecto, … es decir, se ha creado una instantánea del archivo "*archivo_a.txt*" y se ha almacenado en la zona de preparación. Ahora solo nos queda realizar el commit de ese archivo para almacenar esa instantánea de forma segura en el repositorio Git.

4.6 Primer commit

Pues es lo que vamos a hacer, nuestro primer commit, de tal forma que guardaremos en el repositorio una *instantánea* del "*archivo_a.txt*", que por ahora está vacío. Esa instantánea la hemos almacenado previamente en la zona de preparación mediante el comando "*git add*".

Para ello ejecutamos el comando

```
git commit -m "Añado el archivo_a.txt vacío"
```

```
[master (root-commit) bbc294f]  Añado el archivo_a.txt vacío
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 archivo_a.txt
```

Si ahora ejecutamos

```
git status
```

```
On branch master
nothing to commit, working directory clean
```

podemos ver que todo está sincronizado; es decir, que el contenido del directorio de trabajo, el de la zona de preparación y el del repositorio es el mismo.

Como podemos ver, el comando "*git commit*" tiene un parámetro opcional, en el que se le pasa después del parámetro "*-m*" entre comillas dobles el mensaje con el que queremos que se identifique ese commit.

Si necesitamos introducir un mensaje más largo, lo que haremos será ejecutar el comando

```
git commit
```

y automáticamente se abrirá el editor de texto que Git tenga configurado (ver apartado 4.2) o que tengamos como predefinido. Tendremos que escribir el contenido del mensaje del commit, guardar lo escrito y cerrar el editor (si estamos en modo consola) para finalizar el commit.

4.7 Añadiendo más archivos

Ahora vamos a crear y a añadir dos archivos más al proyecto. Para ello ejecutamos

```
touch archivo_b.txt
touch archivo_c.txt
```

Si ahora ejecutamos

```
git status
```

podemos ver que tenemos dos archivos que no se encuentran bajo seguimiento de Git.

```
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

   archivo_b.txt
   archivo_c.txt

nothing added to commit but untracked files present (use "git add" to track)
```

Vamos a pasar el primero de los archivos a ser seguido por Git, a la vez que guardamos una instantánea suya en la zona de preparación. Para ello ejecutamos

```
git add archivo_b.txt
```

Si ahora ejecutamos

```
git status
```

podemos ver que el *archivo_b.txt* está preparado para ser guardado en el repositorio en el próximo commit, mientras que el *archivo_c.txt* sigue sin estar seguido por Git.

```
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

   new file:   archivo_b.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)
   archivo_c.txt
```

Ahora realizamos un commit, de tal forma que almacenamos en el repositorio una instantánea del *archivo_b.txt*, que por ahora está vacío

```
git commit -m "Añado el archivo_b.txt vacío"
```

```
[master a09d278] Añado el archivo_b.txt vacío
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 archivo_b.txt
```

si volvemos a ejecutar

```
git status
```

vemos que ahora solo tenemos el *archivo_c.txt* sin estar bajo seguimiento. Los ficheros *archivo_a.txt* y *archivo_b.txt* se encuentran sincronizados; es decir, que la versión de estos dos archivos en el directorio de trabajo, en la zona de preparación y en repositorio son idénticos, por lo que no aparecen al ejecutar el anterior comando "*git status*".

```
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

   archivo_c.txt

nothing added to commit but untracked files present (use "git add" to track)
```

Solo nos queda por añadir el fichero "*archivo_c.txt*" y ejecutar el commit correspondiente.

Lo primero que hacemos es pasar este archivo a ser seguido por Git, a la vez que guardamos una instantánea suya en la zona de preparación. Para ello ejecutamos el comando

```
git add .
```

En este caso no indicamos explícitamente el nombre del archivo, sino que le indicamos como parámetro ".", que hace referencia al directorio actual, de tal forma que también englobaría a más archivos que no estuvieran bajo seguimiento.

Si volvemos a ejecutar

```
git status
```

vemos que este archivo ya se encuentra en la zona de preparación listo para ser enviado al repositorio mediante un commit

```
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

   new file:   archivo_c.txt
```

Para ello ejecutamos el comando

```
git commit
```

de tal forma que se abre el editor de texto asignado a Git mediante los comandos indicados en el apartado 4.2 o, en su defecto, el asignado al usuario.

En el editor añadimos al inicio del archivo el texto

```
Añado el archivo_c.txt vacío
```

A continuación guardamos y cerramos el editor.

El resultado es

```
1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 archivo_c.txt
```

Si ahora ejecutamos el comando

```
git log --oneline
```

que veremos con detalle en el apartado correspondiente, podemos ver los tres commits realizados, en orden cronológico inverso (primero los más recientes), formado por la versión corta del SHA-1 y por el mensaje del commit

```
e6115ee Añado el archivo_c.txt vacío
a09d278 Añado el archivo_b.txt vacío
bbc294f Añado el primer archivo vacío
```

4.8 Editando archivos

Tras añadir archivos a un proyecto lo habitual es trabajar con ellos, de tal forma que lo que hacemos es editarlos con un editor de texto, un IDE, ... y almacenar las instantáneas de estas

modificaciones en el repositorio mediante los commits correspondientes, para poder tener una trazabilidad de nuestro trabajo.

Para simular la edición del archivo "archivo_a.txt" ejecutamos el comando

```
echo "Creo una primera línea en archivo_a.txt" >> archivo_a.txt
```

Si ejecutamos el comando

```
git status
```

```
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

   modified:   archivo_a.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

Vemos que Git detecta el cambio que acabamos de introducir en el archivo archivo_a.txt; es decir, la versión del archivo existente en la zona de trabajo es diferente a la existente en la zona de preparación y en el repositorio.

Si ahora ejecutamos

```
git add archivo_a.txt
```

Lo que estamos haciendo es almacenar la instantánea del archivo archivo_a.txt en la zona de preparación.

Si ahora volvemos a ejecutar

```
git status
```

Vemos que el archivo está listo para ser enviado al repositorio mediante un commit, por lo que ahora el contenido de la zona de trabajo y de la zona de preparación es idéntico, difiriendo del contenido del repositorio.

```
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

   modified:   archivo_a.txt
```

Lo siguiente que vamos a hacer es editar los otros dos archivos archivo_b.txt y archivo_c.txt, simulándolo mediante el comando "*echo*".

```
echo "Creo una primera línea en el archivo_b.txt" >> archivo_b.txt
echo "Creo una primera línea en el archivo_c.txt" >> archivo_c.txt
```

Si volvemos a ejecutar

```
git status
```

Vemos que seguimos teniendo el archivo archivo_a.txt listo para ser enviado al respositorio, pero ahora tenemos cambios en los archivos archivo_b.txt y archivo_c.txt que aún no han sido enviados ni a la zona de preparación ni al repositorio.

```
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

   modified:   archivo_a.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

   modified:   archivo_b.txt
   modified:   archivo_c.txt
```

Añadimos el archivo *archivo_b.txt* a la zona de preparación

```
git add archivo_b.txt
```

Con lo que si volvemos a comprobar el estado del repositorio mediante el comando

```
git status
```

Vemos que ahora tenemos los archivos *archivo_a.txt* y *archivo_b.txt* listos para enviar al repositorio.

```
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

   modified:   archivo_a.txt
   modified:   archivo_b.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
```

```
        modified:   archivo_c.txt
```

Lo siguiente que vamos a hacer es guardar la instantánea de los archivos que están preparados en el repositorio, mediante el comando

```
git commit -m "Introduzco una línea en los archivos archivo_a.txt y archivo_b.txt"
```

```
[master 689e5d6] Introduzco una línea en los archivos archivo_a.txt y archivo_b.txt
 2 files changed, 2 insertions(+)
```

Si volvemos a ver el estado del repositorio

```
git status
```

Vemos que solo tenemos un archivo en el que hay alguna diferencia entre las tres zonas de Git: *archivo_c.txt*

```
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   archivo_c.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

Si ejecutamos el comando

```
git log --oneline
```

podemos ver el nuevo commit que acabamos de introducir

```
689e5d6 Introduzco una línea en los archivos archivo_a.txt y archivo_b.txt
e6115ee Añado el archivo_c.txt vacío
a09d278 Añado el archivo_b.txt vacío
bbc294f Añado el primer archivo vacío
```

Para finalizar este apartado vamos a añadir el archivo archivo_c.txt al repositorio, mediante los comandos

```
git add archivo_c.txt
```

```
git commit -m "Introduzco una línea en el archivo archivo_c.txt"
```

```
[master b73d0fc] Introduzco una línea en el archivo archivo_c.txt
 1 file changed, 1 insertion(+)
```

Al volver a ejecutar

```
git status
```

Vemos que las tres zonas de trabajo de Git están sincronizadas; es decir, los últimos cambios han sido almacenados en el repositorio.

```
On branch master
nothing to commit, working directory clean
```

Ejecutando

```
git log --oneline
```

vemos que ahora ya tenemos 5 commits

```
b73d0fc Introduzco una línea en el archivo archivo_c.txt
689e5d6 Introduzco una línea en los archivos archivo_a.txt y archivo_b.txt
e6115ee Añado el archivo_c.txt vacío
a09d278 Añado el archivo_b.txt vacío
bbc294f Añado el primer archivo vacío
```

4.9 Borrando archivos

Una tarea bastante habitual es tener que borrar archivos a medida que vamos trabajando en un proyecto. Con Git lo podemos gestionar de dos formas, como vamos a ver a continuación.

Antes de nada vamos a crear dos archivos temporales, que añadiremos al repositorio para borrarlos posteriormente.

```
touch temporal_1.txt
touch temporal_2.txt
```

Si ejecutamos

```
git status
```

Podemos ver que tenemos dos archivos que no están bajo seguimiento.

```
# On branch master
```

```
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#  temporal_1.txt
#  temporal_2.txt
nothing added to commit but untracked files present (use "git add" to track)
```

Los ponemos bajo seguimiento y los pasamos a la zona de preparación

```
git add .
```

Ejecutanto

```
git status
```

Vemos como ahora están listos para realizar el commit.

```
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#  new file:   temporal_1.txt
#  new file:   temporal_2.txt
```

Realizamos el commit

```
git commit -m "Añado dos archivos de prueba para borrar"
```

```
[master f27eb64] Añado dos archivos de prueba para borrar
 2 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 temporal_1.txt
 create mode 100644 temporal_2.txt
```

Lo siguiente que vamos a hacer es borrar un archivo mediante el comando “rm”

```
rm temporal_1.txt
```

Si volvemos a ejecutar

```
git status
```

```
# On branch master
```

```
# Changes not staged for commit:
#   (use "git add/rm <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#  deleted:    temporal_1.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Vemos que nos indica que usemos "git add/rm file… " para actualizar los cambios que serán guardados en el próximo commit, a la vez que nos indica que se ha borrado un archivo.

Ejecutamos

```
git rm temporal_1.txt
```

```
rm 'temporal_1.txt'
```

Y al volver a ejecutar

```
git status
```

```
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#  deleted:    temporal_1.txt
```

Podemos ver que ahora sí aparece el cambio listo para ser llevado al repositorio mediante un commit. Solo nos queda llevar a cabo el commit

```
git commit -m "Borro el archivo temporal_1.txt"
```

```
[master cb9d7bb] Borro el archivo temporal_1.txt
 1 files changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 temporal_1.txt
```

La otra forma que tenemos de borrar el archivo, mucho más rápida y cómoda, es hacerlo directamente mediante el comando "*git rm*".

Ejecutamos

```
git status
```

```
# On branch master
nothing to commit (working directory clean)
```

Para ver que todo está sincronizado

Luego ejecutamos el comando para llevar a cabo el borrado

```
git rm temporal_2.txt
```

```
rm 'temporal_2.txt'
```

Si volvemos a ver el estado del proyecto

```
git status
```

```
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#  deleted:    temporal_2.txt
```

Vemos que solo tenemos que hacer el commit para guardar este cambio en el repositorio

```
git commit -m "Borro el archivo temporal_2.txt"
```

```
[master 0357c83] Borro el archivo temporal_2.txt
 1 files changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 temporal_2.txt
```

Por lo tanto, hemos borrado los archivos de dos formas distintas:

- En la primera borramos el archivo del sistema de ficheros y luego lo borramos en Git.
- En la segunda lo borramos directamente de Git, que provoca su borrado también en la zona de trabajo y en la zona de preparación, quedando listo para realizar el commit.

4.10 Renombrando y moviendo archivos

Lo siguiente que vamos a ver es cómo se renombran y mueven archivos en Git. Un renombrado en un sistema Unix no deja de pasar por tener que mover el archivo desde el nombre antiguo al actual, por lo que usaremos el mismo comando para ambos.

Antes de nada vamos a crear tres archivos temporales, que añadiremos al repositorio para borrarlos posteriormente.

```
touch temporal_3.txt
touch temporal_4.txt
touch temporal_5.txt
```

Si ejecutamos

```
git status
```

Podemos ver que tenemos tres archivos que no están bajo seguimiento.

```
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#  temporal_3.txt
#  temporal_4.txt
#  temporal_5.txt
nothing added to commit but untracked files present (use "git add" to track)
```

Los ponemos bajo seguimiento y los pasamos a la zona de preparación

```
git add .
```

Ejecutando

```
git status
```

Vemos como ahora están listos para realizar el commit.

```
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#  new file:   temporal_3.txt
#  new file:   temporal_4.txt
#  new file:   temporal_5.txt
```

Realizamos el commit

```
git commit -m "Añado tres archivos de prueba para moverlos"
```

```
[master 71ad20c] Añado tres archivos de prueba para moverlos
 3 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 temporal_3.txt
 create mode 100644 temporal_4.txt
 create mode 100644 temporal_5.txt
```

Y lo siguiente que vamos a hacer es mover un archivo mediante el comando "*mv*".

```
mv temporal_3.txt temporal_3_movido.txt
```

Al ejecutar

```
git status
```

```
# On branch master
# Changes not staged for commit:
#   (use "git add/rm <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#  deleted:    temporal_3.txt
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#  temporal_3_movido.txt
no changes added to commit (use "git add" and/or "git commit -a")
```

Podemos ver que Git detecta que se ha eliminado un archivo y que hay otro que no se encuentra bajo seguimiento. Además nos indica que tenemos que ejecutar "git add/rm <file>…" para actualizar los cambios a guardar en el repositorio.

Ejecuto

```
git rm temporal_3.txt
git add temporal_3_movido.txt
```

Y al volver a ejecutar

```
git status
```

```
# On branch master
```

```
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#  renamed:    temporal_3.txt -> temporal_3_movido.txt
```

Git se da cuenta de que lo que realmente acabamos de hacer es renombrar el archivo "*temporal_3.txt*" a "*temporal_3_movido.txt*".

La otra forma que tenemos para mover un archivo es utilizar el comando "*git mv*", que va a mover o renombrar el archivo en el sistema de archivos.

```
git mv temporal_4.txt temporal_4_movido.txt
```

Si ahora ejecutamos

```
git status
```

Aparecen los dos archivos renombrados

```
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#  renamed:    temporal_3.txt -> temporal_3_movido.txt
#  renamed:    temporal_4.txt -> temporal_4_movido.txt
```

Ahora vamos a mover un archivo de ubicación. Para ello creamos un directorio temporal, ejecutando

```
mkdir dir_temp
```

Y para mover el archivo usamos el comando "*git mv*".

```
git mv temporal_5.txt dir_temp/temporal_5.txt
```

Al volver a ejecutar

```
git status
```

Aparecen los tres archivos renombrados

```
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#  renamed:    temporal_5.txt -> dir_temp/temporal_5.txt
#  renamed:    temporal_3.txt -> temporal_3_movido.txt
```

```
        #  renamed:    temporal_4.txt -> temporal_4_movido.txt
```

Ahora solo nos queda llevar a cabo el commit:

```
git commit -m "Renombramos tres archivos"
```

```
[master 84d4898] Renombramos tres archivos
 3 files changed, 0 insertions(+), 0 deletions(-)
 rename temporal_5.txt => dir_temp/temporal_5.txt (100%)
 rename temporal_3.txt => temporal_3_movido.txt (100%)
 rename temporal_4.txt => temporal_4_movido.txt (100%)
```

4.11 Ignorar archivos

De vez en cuando necesitamos excluir ciertos archivos del seguimiento que lleva a cabo Git. Puede que si estamos creando software solo nos interese tener el código fuente bajo seguimiento y no los archivos propios de la compilación (.slo, .lo, .o, .obj), librerías compiladas (.so, .dylib, .dll, .lai, .la, .a, .lib) o los propios ejecutables (.exe, .out, .app).

Para ello lo que haremos será crear un archivo ".gitignore" en el directorio raíz del proyecto que se encuentra bajo seguimiento de Git y luego en cada línea introduciremos un patrón para que sea Git el que decida si excluye o no un determinado elemento.

El formato de los patrones tiene en cuenta los siguientes elementos:

- Una línea en blanco es un separador para mejorar la legibilidad del archivo.
- Los comentarios empiezan con el carácter "#". Para poder escapar este carácter y poder usarlo pondremos una barra invertida "\" antes de la almohadilla: \#.
- El prefijo "!" niega el patrón. Si un patrón excluye un archivo determinado esta regla podrá volver a incluirlo. Para poder escapar este carácter y poder usarlo pondremos una barra invertida "\" antes de la almohadilla: \!.
- Podemos usar el carácter "*" para reemplazar a un número indeterminado de caracteres.
- Podemos usar el carácter "?" para reemplazar un carácter cualquiera.
- Podemos usar expresiones regulares del tipo [0-9], que sustituye a un número entre el 0 y el 9 o [oa], que sustituye a la letra "a" o la "o".

Veamos todo esto con un ejemplo.

Para empezar añadimos varios directorios, archivos y el archivo .gitignore:

```
touch temporal_6.txt
touch temporal_7.zip
mkdir log
touch log/apache.log
```

```
touch log/apache.log1
mkdir imagenes
touch imagenes/fondo.png
touch imagenes/logo.png
mkdir compilados
touch compilados/salida.o
touch compilados/salida.a
touch .gitignore
```

Si ejecutamos

```
git status
```

Vemos que los archivos y directorios no están bajo seguimiento.

```
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#  .gitignore
#  compilados/
#  imagenes/
#  log/
#  temporal_6.txt
#  temporal_7.zip
nothing added to commit but untracked files present (use "git add" to track)
```

Añado las líneas correspondientes al archivo .gitignore mediante el comando “echo”.

```
# Añadimos el archivo "temporal_6.txt" y el "temporal_7.zip"
echo "temporal_6.txt" >> .gitignore
echo "temporal_7.zip" >> .gitignore
# Añadimos todos los archivos .zip y .gz en el directorio raíz
echo "*.zip" >> .gitignore
echo "*.gz" >> .gitignore
# Añadimos los archivos de log que están en el directorio "log"
echo "log/*.log" >> .gitignore
# Añadimos los archivos de rotación de log que están en el
directorio "log"
echo "log/*.log[0-9]" >> .gitignore
```

```
# Añadimos todo el contenido del directorio "imagenes"
echo "imagenes/*" >> .gitignore
# Excepto el archivo "logo.png"
echo "!imagenes/logo.png" >> .gitignore
# Añadimos los archivos compilados
echo "compilados/*[ao]" >> .gitignore
```

Si ejecutamos

```
git status
```

Vemos que ahora solo aparece el archivo .gitignore.

```
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#  .gitignore
# imagenes/
nothing added to commit but untracked files present (use "git add" to track)
```

Solo nos queda por añadir los archivos "imagenes/logo.png" y ".gitignore" al repositorio. Para ello ejecutamos

```
git add .
git commit -m "Añado el archivo .gitignore y el archivo
imagenes/logo.png"
```

```
[master 69c7d78] Añado el archivo .gitignore y el archivo imagenes/logo.png
 2 files changed, 9 insertions(+)
 create mode 100644 .gitignore
 create mode 100644 imagenes/logo.png
```

4.12 Deshacer cambios: zona de trabajo y zona de preparación

Lo siguiente que vamos a ver es cómo se deshacen los cambios en la zona de trabajo y en la zona de preparación.

Para ello introducimos un cambio en el archivo_a.txt

```
echo "Creo una segunda línea en archivo_a.txt" >> archivo_a.txt
git status
```

```
En la rama master
Cambios no preparados para el commit:
  (use «git add <archivo>...» para actualizar lo que se ejecutará)
  (use «git checkout -- <archivo>...« para descartar cambios en le directorio
de trabajo)

   modified:   archivo_a.txt

no hay cambios agregados al commit (use «git add» o «git commit -a»)
```

En la salida del comando "git status" nos indica que ejecutemos "git checkout -- <archivo>" para descartar los cambios en el directorio de trabajo.

```
git checkout -- archivo_a.txt
git status
```

```
En la rama master
nothing to commit, working directory clean
```

Y ya volvemos a estar en la situación de partida, sin que nos afecten los cambios.

A continuación vamos a ver cómo hacemos lo mismo pero una vez que hemos añadido al índice o zona de preparación el archivo.

```
echo "Creo una segunda línea en archivo_a.txt" >> archivo_a.txt
git add archivo_a.txt
git status
```

```
En la rama master
Cambios para hacer commit:
  (use «git reset HEAD <archivo>...«para eliminar stage)

   modified:   archivo_a.txt
```

En la salida del comando "git status" nos indica que ejecutemos "git reset HEAD <archivo>" para descartar los cambios en la zona de preparación.

```
git reset HEAD archivo_a.txt
```

```
Unstaged changes after reset:
M  archivo_a.txt
```

```
git status
```

```
En la rama master
Cambios no preparados para el commit:
  (use «git add <archivo>...» para actualizar lo que se ejecutará)
  (use «git checkout -- <archivo>...« para descartar cambios en le directorio de trabajo)

   modified:   archivo_a.txt

no hay cambios agregados al commit (use «git add» o «git commit -a»)
```

Para volver a la situación de partida volvemos a ejecutar el comando "git checkout -- <archivo>" para descartar los cambios en el directorio de trabajo.

```
git checkout -- archivo_a.txt
git status
```

```
En la rama master
nothing to commit, working directory clean
```

4.13 Modificar commits

Dado que el SHA-1 de un commit depende de los predecesores, solo vamos a poder modificar el último commit.

```
git log --oneline -4
```

```
fd79ed5 Añado el archivo .gitignore y el archivo imagenes/logo.png
44a9a8c Renombramos tres archivos
63a453b Añado tres archivos de prueba para moverlos
b5b78cc Borro el archivo temporal_2.txt
```

Añadimos una línea al archivo .gitignore

```
echo "ejecutables/*.exe" >> .gitignore
git status
```

```
En la rama master
Cambios no preparados para el commit:
  (use «git add <archivo>...» para actualizar lo que se ejecutará)
  (use «git checkout -- <archivo>...« para descartar cambios en le directorio de trabajo)

	modified:   .gitignore

no hay cambios agregados al commit (use «git add» o «git commit -a»)
```

Y lo que hacemos es un nuevo commit, pero con el parámetro "--amend", cambiando también el mensaje del commit.

```
git add .gitignore
git commit --amend -m "Añado los archivos .gitignore e imagenes/logo.png"
```

```
[master f00704b] Añado los archivos .gitignore e imagenes/logo.png
 2 files changed, 10 insertions(+)
 create mode 100644 .gitignore
 create mode 100644 imagenes/logo.png
```

Si volvemos a ver el historial, vemos como se ha cambiado el mensaje del último commit y su SHA-1.

```
git log --oneline -4
```

```
f00704b Añado los archivos .gitignore e imagenes/logo.png
44a9a8c Renombramos tres archivos
63a453b Añado tres archivos de prueba para moverlos
b5b78cc Borro el archivo temporal_2.txt
```

Si vemos el contenido del archivo .gitignore vemos la línea añadida en la última línea.

```
cat .gitignore
```

```
temporal_6.txt
temporal_7.zip
*.zip
*.gz
log/*.log
log/*.log[0-9]
imagenes/*
!imagenes/logo.png
compilados/*[ao]
ejecutables/*.exe
```

4.14 Revertir el estado a un commit

Este comando genera un nuevo commit que deshace los cambios introducidos en el commit indicado, aplicando los cambios a la rama actual.

```
git log --oneline -4
```

Vamos a revertir el último commit realizado.

```
f00704b Añado los archivos .gitignore e imagenes/logo.png
44a9a8c Renombramos tres archivos
63a453b Añado tres archivos de prueba para moverlos
b5b78cc Borro el archivo temporal_2.txt
```

```
git revert f00704b
```

```
Revert "Añado los archivos .gitignore e imagenes/logo.png"
```

```
This reverts commit f00704b429685023339a927f3d0859ffee64d709.

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# En la rama master
# Cambios para hacer commit:
#       deleted:    .gitignore
#       deleted:    imagenes/logo.png
#
```

```
# Archivos sin seguimiento:
#       compilados/
#       imagenes/
#       log/
#       temporal_6.txt
#       temporal_7.zip
#
```

```
[master 7c51cef] Revert "Añado los archivos .gitignore e imagenes/logo.png"
 2 files changed, 10 deletions(-)
 delete mode 100644 .gitignore
 delete mode 100644 imagenes/logo.png
```

Si volvemos a ver el historial, vemos que aparece un nuevo commit.

```
git log --oneline -4
```

```
7c51cef Revert "Añado los archivos .gitignore e imagenes/logo.png"
f00704b Añado los archivos .gitignore e imagenes/logo.png
44a9a8c Renombramos tres archivos
63a453b Añado tres archivos de prueba para moverlos
```

Si vemos el estado del repositorio, como en este último commit se ha eliminado el archivo .gitignore, van a aparecer unos cuantos archivos que se han creado previamente pero que eran ignorados por el repositorio gracias al archivo .gitignore.

```
git status
```

```
En la rama master
Archivos sin seguimiento:
  (use «git add <archivo>...» para incluir lo que se ha de ejecutar)

    compilados/
    imagenes/
    log/
    temporal_6.txt
    temporal_7.zip
```

```
no se ha agregado nada al commit pero existen archivos sin seguimiento (use
«git add» para darle seguimiento)
```

4.15 Eliminar archivos no seguidos

Lo que vamos a hacer a continuación es eliminar los archivos no seguidos en la zona de trabajo.

```
git status
```

```
En la rama master
Archivos sin seguimiento:
  (use «git add <archivo>...» para incluir lo que se ha de ejecutar)

   compilados/
   imagenes/
   log/
   temporal_6.txt
   temporal_7.zip

no se ha agregado nada al commit pero existen archivos sin seguimiento (use
«git add» para darle seguimiento)
```

Vemos que hay varios archivos que no están bajo seguimiento.

Para eliminarlos ejecuto el comando

```
git clean
```

```
fatal: clean.requireForce defaults to true and neither -i, -n, nor -f given;
refusing to clean
```

Pero requiere el parámetro "-f" para que sea efectivo.

```
git clean -f
```

```
Eliminando temporal_6.txt
Eliminando temporal_7.zip
```

Ha borrado 2 archivos. Si vemos el estado del repositorio vemos que los archivos que se encuentran dentro de directorios no han sido eliminados

```
git status
```

```
En la rama master
Archivos sin seguimiento:
  (use «git add <archivo>...» para incluir lo que se ha de ejecutar)

   compilados/
   imagenes/
   log/

no se ha agregado nada al commit pero existen archivos sin seguimiento (use
«git add» para darle seguimiento)
```

Para ello tenemos que usar el parámetro "-d".

```
git clean -f -d
```

```
Eliminando compilados/
Eliminando imagenes/
Eliminando log/
```

Como podemos ver, ahora el repositorio está limpio.

```
git status
```

```
En la rama master
nothing to commit, working directory clean
```

Si vemos qué contiene el directorio de trabajo, podemos apreciar que estos archivos han sido eliminados.

```
ls -1
```

```
archivo_a.txt
archivo_b.txt
archivo_c.txt
dir_temp
temporal_3_movido.txt
```

```
temporal_4_movido.txt
```

4.16 Revertir un archivo de un commit

En determinados casos nos interesa recuperar un archivo de un commit determinado. Por ejemplo, si sabemos dónde se ha introducido un bug, podemos recuperar el archivo del commit anterior. Vamos a hacer esto con el anterior archivo .gitignore.

```
git log --oneline -4
```

```
7c51cef Revert "Añado los archivos .gitignore e imagenes/logo.png"
f00704b Añado los archivos .gitignore e imagenes/logo.png
44a9a8c Renombramos tres archivos
63a453b Añado tres archivos de prueba para moverlos
```

Para recuperarlo, usamos el comando "git checkout" pasándole como parámetros el SHA-1 del commit y el archivo a recuperar precedido de "--".

```
git checkout f00704b -- .gitignore
git status
```

```
En la rama master
Cambios para hacer commit:
  (use «git reset HEAD <archivo>...«para eliminar stage)

   new file:   .gitignore
```

Vemos que lo deja en la zona de preparación, que ya está listo para llevar a cabo el commit.

```
ls -a -1
```

Y si vemos el contenido de la zona de trabajo podemos ver el archivo .gitignore.

```
archivo_a.txt
archivo_b.txt
archivo_c.txt
dir_temp
.git
.gitignore
temporal_3_movido.txt
temporal_4_movido.txt
```

Y si vemos su contenido podemos ver que es correcto.

```
cat .gitignore
```

```
temporal_6.txt
temporal_7.zip
*.zip
*.gz
log/*.log
log/*.log[0-9]
imagenes/*
!imagenes/logo.png
compilados/*[ao]
ejecutables/*.exe
```

Para finalizar realizamos el commit

```
git commit -am "Añado el archivo .gitignore"
```

```
[master 87a4cd0] Añado el archivo .gitignore
 1 file changed, 10 insertions(+)
 create mode 100644 .gitignore
```

4.17 Deshaciendo commit mediante reset: soft, mixed, hard

Este es uno de los pocos comandos con los que podemos perder información en Git, por lo que lo hay que usar con precaución.

4.17.1 Hard reset

Vemos cuál es el estado del repositorio

```
git log --oneline -4
```

```
87a4cd0 Añado el archivo .gitignore
7c51cef Revert "Añado los archivos .gitignore e imagenes/logo.png"
f00704b Añado los archivos .gitignore e imagenes/logo.png
44a9a8c Renombramos tres archivos
```

Si ejecutamos el comando "git reset" con el parámetro "--hard" seguido de un SHA-1 determinado, lo que vamos a hacer es que tanto el repositorio como la zona de preparación y la zona de trabajo cambien al contenido de ese commit.

```
git reset --hard f00704b
```

```
HEAD is now at f00704b Añado los archivos .gitignore e imagenes/logo.png
```

```
git log --oneline -4
```

```
f00704b Añado los archivos .gitignore e imagenes/logo.png
44a9a8c Renombramos tres archivos
63a453b Añado tres archivos de prueba para moverlos
b5b78cc Borro el archivo temporal_2.txt
```

```
git status
```

```
En la rama master
nothing to commit, working directory clean
```

4.17.2 Mixed

Si ejecutamos el comando "git reset" con el parámetro "--mixed" seguido de un SHA-1 determinado, se cambia el contenido del repositorio y de la zona de preparación al estado de ese commit, pero no la de la zona de trabajo.

Volvemos al commit inicial

```
git reset --hard 87a4cd0
```

```
HEAD is now at 87a4cd0 Añado el archivo .gitignore
```

Si vemos los últimos 4 commits

```
git log --oneline -4
```

```
87a4cd0 Añado el archivo .gitignore
7c51cef Revert "Añado los archivos .gitignore e imagenes/logo.png"
f00704b Añado los archivos .gitignore e imagenes/logo.png
```

```
44a9a8c Renombramos tres archivos
```

y la diferencia entre el antepenúltimo y el último

```
git diff f00704b..87a4cd0
```

```
diff --git a/imagenes/logo.png b/imagenes/logo.png
deleted file mode 100644
index e69de29..0000000
```

Vemos que se ha borrado un archivo, el logo.

Realizamos ahora el reset

```
git reset --mixed f00704b
```

```
Unstaged changes after reset:
D  imagenes/logo.png
```

Y si vemos el estado del repositorio vemos que tenemos el cambio anterior, pero que por ahora se encuentra en el directorio de trabajo, no en la zona de preparación ni en el repositorio.

```
git status
```

```
En la rama master
Cambios no preparados para el commit:
  (use «git add/rm <archivo>...» para actualizar lo que se ejecutará)
  (use «git checkout -- <archivo>...« para descartar cambios en le directorio de trabajo)

   deleted:    imagenes/logo.png

no hay cambios agregados al commit (use «git add» o «git commit -a»)
```

```
git log --oneline -4
```

Sin embargo, si vemos los últimos 4 commits, vemos que han desaparecido los 2 últimos

```
f00704b Añado los archivos .gitignore e imagenes/logo.png
44a9a8c Renombramos tres archivos
```

```
63a453b Añado tres archivos de prueba para moverlos
b5b78cc Borro el archivo temporal_2.txt
```

Si vemos las diferencias entre la zona de trabajo y la de preparación es precisamente ese archivo.

```
git diff
```

```
diff --git a/imagenes/logo.png b/imagenes/logo.png
deleted file mode 100644
index e69de29..0000000
```

Sin embargo, si vemos las diferencias entre la zona de preparación y el repositorio, no hay ninguna.

```
git diff --staged
```

```

```

4.17.3 Soft

Si ejecutamos el comando "git reset" con el parámetro "--soft" seguido de un SHA-1 determinado, se cambia el contenido del repositorio al estado de ese commit, pero no la de la zona de preparación ni la de trabajo.

Volvemos al commit inicial

```
git reset --hard 87a4cd0
```

```
HEAD is now at 87a4cd0 Añado el archivo .gitignore
```

```
git log --oneline -4
```

Si vemos los últimos 4 commits

```
87a4cd0 Añado el archivo .gitignore
7c51cef Revert "Añado los archivos .gitignore e imagenes/logo.png"
f00704b Añado los archivos .gitignore e imagenes/logo.png
44a9a8c Renombramos tres archivos
```

y la diferencia entre el antepenúltimo y el último

```
git diff f00704b..87a4cd0
```

```
diff --git a/imagenes/logo.png b/imagenes/logo.png
deleted file mode 100644
index e69de29..0000000
```

Vemos que se ha borrado un archivo, el logo.

Realizamos ahora el reset

```
git reset --soft f00704b
git status
```

```
En la rama master
Cambios para hacer commit:
  (use «git reset HEAD <archivo>...«para eliminar stage)

   deleted:    imagenes/logo.png
```

No hay ninguna diferencia entre la zona de trabajo y la zona de preparación.

```
git diff
```

```

```

Si embargo, sí que la hay entre la zona de preparación y el repositorio.

```
git diff --staged
```

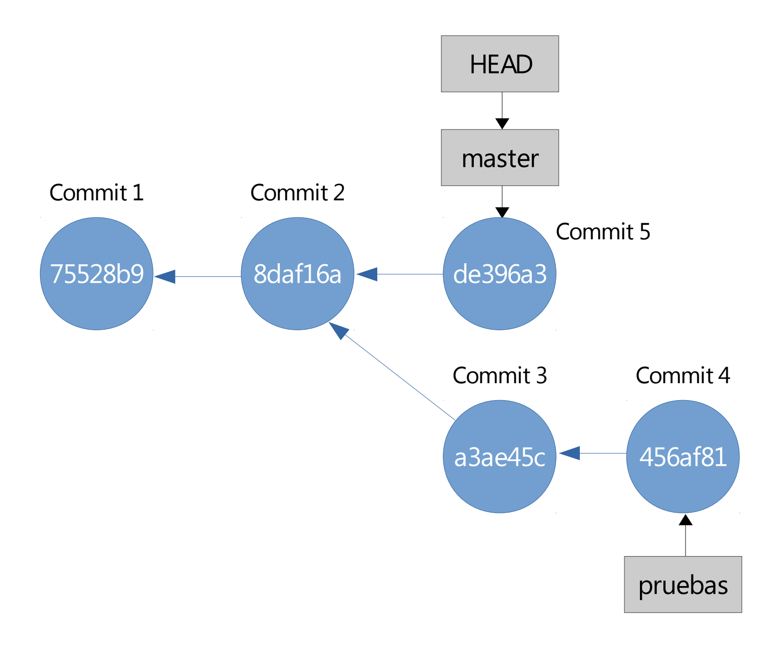
```
diff --git a/imagenes/logo.png b/imagenes/logo.png
deleted file mode 100644
index e69de29..0000000
```

5 Ramas

Cuando estamos trabajando en un proyecto software, en un desarrollo web, en un diseño, ... llega un momento en el que tenemos una versión que publicamos, que podemos llamar estable o de producción, pero necesitamos seguir trabajando en nuevas funcionalidades. Otras veces queremos estudiar nuevas funcionalidades que puede que sean interesantes o puede que tengamos que descartar, probar ideas que pueden ser fructíferas, ... Para esto y para mucho más utilizamos las ramas.

Una rama (branch) es una bifurcación o camino alternativo que seguimos en un determinado momento.

Git, por defecto, crea una rama denominada *master* cuando inicializamos el repositorio, sobre la que empezamos a realizar el trabajo; pero podemos crear más ramas y cambiarnos entre ellas, de tal forma que el contenido del directorio de trabajo irá cambiando.



En la figura anterior podemos ver un ejemplo en el que:

- Hemos creado dos commits (75528b9 y 8daf16a) en la rama principal, *master*.
- A continuación hemos creado una nueva rama, *pruebas*, en la que hemos introducido dos nuevos commits (a3ae45c y 456af81).
- Luego nos hemos cambiado a la rama *master*, y hemos introducido un nuevo commit (de396a3).
- Podemos ver que HEAD apunta al último commit de la rama master, que es la rama activa.

HEAD siempre apunta al último commit de la rama que está activa; es decir, de la que tenemos los contenidos en el directorio de trabajo.

5.1 Por qué usar una rama

Los motivos para utilizar una rama o camino alternativo son muy variados, pero siempre muy útiles. Entre los más habituales podemos encontrar:

- Tener la versión estable en la rama *master* y tener otra rama, la de "*desarrollo*" para seguir trabajando en funcionalidades que no están tan probadas. Incluso podemos tener varias ramas, en función del nivel de estabilidad del código: ramas *alfa* y *beta*.
- Poder probar ideas que se nos ocurren y que puede que tengamos que descartar, pero si son adecuadas puede que pasen a la rama principal de desarrollo en un futuro cercano.
- Arreglar errores (*bugs*), de tal forma que podemos llevar a cabo las pruebas en una rama específica para ese error y si todo es correcto podemos llevar los cambios a la rama principal y publicar una actualización.

5.2 Crear una rama

Para ver cómo trabajamos con ramas vamos a crear un proyecto nuevo con dos commits, uno para cada archivo nuevo creado, mediante el siguiente script:

```
cd ~/proyectos/
mkdir proyecto_ramas
cd proyecto_ramas
git init
touch archivo_a.txt
git add .
git commit -m "Añado el primer archivo"
touch archivo_b.txt
git add .
git commit -m "Añado el segundo archivo"
```

Podemos ver el historial de commits

```
git log --oneline
```

```
1a9f1c2 Añado el segundo archivo
eff327b Añado el primer archivo
```

Si ejecutamos

```
git branch
```

Podemos ver las ramas que tenemos creadas.

```
* master
```

Por ahora solo tenemos la rama *master*, que crea Git por defecto cuando inicializamos el repositorio.

Si ejecutamos

```
ls -a -1 .git
```

Podemos ver que hay un archivo con el nombre HEAD.

```
branches
COMMIT_EDITMSG
config
description
HEAD
hooks
index
info
logs
objects
refs
```

Si vemos su contenido mediante el comando

```
cat .git/HEAD
```

Podemos ver que muestra una referencia hacia /refs/heads/master

```
ref: refs/heads/master
```

Si mostramos lo que hay en el directorio “/refs/heads/”

```
ls -a -1 .git/refs/heads/
```

Vemos que por tenemos ese archivo master referenciado desde el contenido de HEAD

```
master
```

Si mostramos su contenido con el comando

```
cat .git/refs/heads/master
```

Podemos ver que aparece el SHA-1 del último commit de la rama master.

```
1a9f1c256aa7aab787fbc0a0d95b971025a13490
```

Ahora vamos a crear una nueva rama, mediante el comando branch seguido del nombre que le queremos dar a la rama

```
git branch pruebas
```

Lo que acabamos de hacer es crear una rama llamada "*pruebas*", pero aún seguimos trabajando en la rama master, ya que si ejecutamos

```
git branch
```

```
* master
  pruebas
```

Podemos ver que la rama que tiene un "*" en la parte izquierda es la rama *master*. El "*" nos indica cuál es la rama activa.

Si ahora volvemos a ver el contenido

```
ls -la .git/refs/heads/
```

```
master
pruebas
```

Vemos que acaba de aparecer un nuevo archivo, "pruebas", que incluirá el SHA-1 del último commit de esta rama.

Si lo visualizamos mediante el comando

```
cat .git/refs/heads/pruebas
```

```
1a9f1c256aa7aab787fbc0a0d95b971025a13490
```

Y hacemos lo mismo con el archivo de la rama master

```
cat .git/refs/heads/master
```

```
1a9f1c256aa7aab787fbc0a0d95b971025a13490
```

Vemos que contienen lo mismo, ya que por ahora ambas ramas apuntan al mismo commit. Cuando introduzcamos un commit en una de las dos ramas este contenido cambiará.

5.3 Mostrar las ramas

Para mostrar todas las ramas ejecutamos el comando

```
git branch
```

```
* master
```

```
    pruebas
```

Si estuviéramos trabajando con un servidor remoto, para poder mostrar las ramas de seguimiento usaremos el comando

```
git branch -r
```

Y para mostrar todas las ramas usaremos el comando

```
git branch -a
```

```
  * master
    pruebas
```

5.4 Cambiarse de rama

Si queremos cambiarnos de rama para continuar trabajando en otra rama utilizaremos el comando "*git checkout*".

Si ejecutamos

```
git branch
```

Vemos que estamos en la rama master

```
  * master
    pruebas
```

Antes de cambiarnos de rama vemos el contenido del HEAD

```
cat .git/HEAD
```

```
  ref: refs/heads/master
```

Y vemos que apunta a la rama "master".

Para cambiarnos de rama activa de trabajo desde la rama master a la rama de pruebas ejecutamos

```
git checkout pruebas
```

```
  Switched to branch 'pruebas'
```

Si ejecutamos

```
git branch
```

Vemos que ahora estamos en la rama pruebas.

```
  master
* pruebas
```

Si vemos el contenido del HEAD

```
cat .git/HEAD
```

Comprobamos que apunta a la rama “pruebas”.

```
ref: refs/heads/pruebas
```

5.5 Trabajando con una rama

A continuación vamos a añadir contenido al archivo “archivo_a.txt” en la rama pruebas. Para ello nos aseguramos de que estamos en la rama mediante

```
git checkout pruebas
```

Y añadimos una línea al archivo indicado

```
echo "Inserto una línea en el archivo_a.txt" >> archivo_a.txt
```

A continuación añado los archivos a la zona de preparación y realizo el commit con un único comando

```
git commit -am "Introduzco una línea en el archivo archivo_a.txt"
```

El comando "git commit -am" es equivalente a git add [archivos] y git commit -m "Mensaje" una vez que los archivos están bajo seguimiento por Git.

```
[pruebas 117d623] Introduzco una línea en el archivo archivo_a.txt
 1 file changed, 1 insertion(+)
```

Si vemos el historial

```
git log --oneline
```

Podemos comprobar que en esta rama tenemos tres commits.

```
117d623 Introduzco una línea en el archivo archivo_a.txt
1a9f1c2 Añado el segundo archivo
eff327b Añado el primer archivo
```

Si vemos el contenido del archivo “*archivo_a.txt*”.

```
cat archivo_a.txt
```

Podemos ver la línea que acabamos de introducir.

```
Inserto una línea en el archivo_a.txt
```

Ahora vamos a cambiarnos a la rama "master". Para ello ejecutamos

```
git checkout master
```

Si vemos el contenido del archivo "archivo_a.txt".

```
cat archivo_a.txt
```

```

```

Podemos ver que la línea que acabamos de introducir ya no está. Esto se debe a que esta línea la hemos introducido en la rama "pruebas", pero no en la rama "*master*". Más adelante veremos como podemos llevar los cambios de una rama a otra.

Si ejecutamos

```
git log --oneline
```

```
1a9f1c2 Añado el segundo archivo
eff327b Añado el primer archivo
```

Vemos que en esta rama solo tenemos dos commits, ya que el último commit ha tenido lugar en la rama "*pruebas*".

Si nos volvemos a cambiar a la rama "*pruebas*"

```
git checkout pruebas
```

Ejecutando

```
git log --oneline
```

Podemos ver que en esa rama tenemos tres commits.

```
117d623 Introduzco una línea en el archivo archivo_a.txt
1a9f1c2 Añado el segundo archivo
eff327b Añado el primer archivo
```

Como podemos ver, el contenido del espacio de trabajo cambia acorde a nuestros cambios y commits en la rama activa.

5.6 Crear y cambiar a una rama

Hasta ahora hemos visto que para crear una rama y empezar a utilizarla tenemos que seguir dos pasos. Lo primero que hacemos es cambiarnos a la rama desde la que queremos crear la nueva rama; por ejemplo, la rama “master”

```
git checkout master
```

Y a continuación creamos la rama con el comando

```
git branch [nombre_de_la_rama]
```

Luego nos cambiamos a esa nueva rama con el comando

```
git checkout [nombre_de_la_rama]
```

Pero tenemos una forma de hacerlo en un único paso, tras cambiarnos a la rama a partir de la que queremos crear la rama nueva, que en el ejemplo es “*master*”

```
git checkout master
git checkout -b experimento
```

```
Switched to a new branch 'experimento'
```

Con este último comando acabamos de crear la rama y nos hemos cambiado a ella. Los cambios que hagamos a partir de ahora quedarán almacenados en la rama “experimento”.

A continuación añadimos una línea al archivo “*archivo_a.txt*”.

```
echo "Experimento añadiendo una nueva línea al archivo_a.txt en la
rama experimento" >> archivo_a.txt
```

Si vemos el estado del repositorio

```
git status
```

```
# On branch experimento
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#  modified:   archivo_a.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

vemos que tenemos el "*archivo_a.txt*" modificado.

A continuación realizamos el commit con el comando

```
git commit -am "Experimentando con una nueva línea en el archivo_a.txt en la rama experimento"
```

```
[experimento 9618f2a] Experimentando con una nueva línea en el archivo_a.txt en la rama experimento
 1 file changed, 1 insertion(+)
```

Si vemos el histórico de commits

```
git log --oneline
```

ahora tenemos un nuevo commit.

```
9618f2a Experimentando con una nueva línea en el archivo_a.txt en la rama experimento
1a9f1c2 Añado el segundo archivo
eff327b Añado el primer archivo
```

Si vemos el contenido del archivo

```
cat archivo_a.txt
```

```
Experimento añadiendo una nueva línea al archivo_a.txt en la rama experimento
```

Ahora nos cambiamos a la rama "master"

```
git checkout master
```

Si vemos el histórico de commits

```
git log --oneline
```

```
1a9f1c2 Añado el segundo archivo
eff327b Añado el primer archivo
```

no tenemos el último commit, ya que se ha llevado a cabo en la rama "experimento".

Si vemos el contenido del archivo

```
cat archivo_a.txt
```

este está vacío, ya que la edición del archivo se ha llevado a cabo en la rama “experimento”.

A continuación nos cambiamos a la rama “pruebas”

```
git checkout pruebas
```

Si vemos el histórico de commits

```
git log --oneline
```

```
117d623 Introduzco una línea en el archivo archivo_a.txt
1a9f1c2 Añado el segundo archivo
eff327b Añado el primer archivo
```

tampoco tenemos el último commit, ya que se ha llevado a cabo en la rama “experimento”.

Si vemos el contenido del archivo

```
cat archivo_a.txt
```

```
Inserto una línea en el archivo_a.txt
```

el contenido es distinto del introducido en la rama “experimento”.

Si ejecutamos

```
git log --oneline --graph --all --decorate
```

```
* 9618f2a (experimento) Experimentando con una nueva línea en el archivo_a.txt
| * 117d623 (HEAD, pruebas) Introduzco una línea en el archivo archivo_a.txt
|/
* 1a9f1c2 (master) Añado el segundo archivo
* eff327b Añado el primer archivo
```

Podemos ver en el gráfico que a partir del commit 1a9f1c2 de la rama “*master*” se han creado dos ramas, “pruebas” y “experimento”, y en cada una de ellas se ha añadido un commit en el que se ha introducido una línea en el archivo “archivo_a.txt”.

La rama activa es “pruebas”, ya que es hacia donde apunta el HEAD.

5.7 Renombrar una rama

Si necesitamos renombrar una rama lo que haremos será utilizar el siguiente comando:

```
git branch -m [nombre_actual] [nombre_nuevo]
```

Antes de empezar el ejemplo vamos a ver el nombre de las ramas que tenemos

```
git branch
```

```
  experimento
  master
* pruebas
```

Si queremos renombrar la rama “experimento” que acabamos de crear con el nombre “intento”, ejecutaremos el comando

```
git branch -m experimento intento
```

Si ahora vemos las ramas que tenemos

```
git branch
```

```
  intento
  master
* pruebas
```

vemos que la rama “experimento” ha sido renombrada con el nombre “intento”.

5.8 Borrar una rama

En muchas ocasiones crearemos ramas de una duración corta para realizar pruebas, arreglar errores, ... Tras utilizar la rama y llevar sus modificaciones a otra rama, lo mejor que podemos hacer es borrar la rama, para tener el repositorio lo más limpio y ordenado posible. Para ello usaremos el siguiente comando

```
git branch -d [nombre_de_la_rama]
```

En este ejemplo vamos a crear una rama nueva, que vamos a borrar a continuación

```
git checkout master
git branch rama_temporal
git checkout rama_temporal
git branch -d rama_temporal
```

```
error: Cannot delete the branch 'rama_temporal' which you are currently on.
```

Como podemos ver, no es posible borrar una rama en la que nos encontramos, por lo que tendremos que cambiarnos de rama

```
git checkout master
git branch -d rama_temporal
```

```
Deleted branch rama_temporal (was 1a9f1c2).
```

5.9 Fusionar ramas

Cuando creamos una rama el objetivo final suele ser incorporar ese trabajo a otra rama distinta, de tal forma que en la rama nueva realizamos pruebas, arreglamos errores, ... y tras comprobar que esos cambios funcionan correctamente los llevamos a otra rama. Para fusionar las ramas usaremos el comando

```
git merge
```

Veamos las ramas que tenemos ahora mismo

```
git branch
```

```
  intento
* master
  pruebas
```

Si vemos como están los commits en las distintas ramas con el comando

```
git log --oneline --graph --all --decorate
```

```
* 9618f2a (intento) Experimentando con una nueva línea en el archivo_a.txt
| * 117d623 (pruebas) Introduzco una línea en el archivo archivo_a.txt
|/
* 1a9f1c2 (HEAD, master) Añado el segundo archivo
* eff327b Añado el primer archivo
```

Vemos que la rama “intento” está un commit por delante de la rama “master”.

Lo que vamos a hacer ahora es llevar los cambios de la rama “intento” a la rama “master”. Para ello, tras ubicarnos en la rama a la que queremos llevar los cambios, ejecutamos el comando “git merge”, al que le pasaremos como parámetro la rama que queremos fusionar en la rama activa.

```
git checkout master
git merge intento
```

```
Updating 1a9f1c2..9618f2a
Fast-forward
 archivo_a.txt |    1 +
 1 file changed, 1 insertion(+)
```

Vemos que indica que se ha hecho un "*fast-forward*", ya que lo único que hemos tenido que hacer en la rama "master" es avanzar un commit siguiendo el camino de la rama "intento".

Si ejecutamos

```
git log --oneline --graph --all --decorate
```

```
* 9618f2a (HEAD, master, intento) Experimentando con una nueva línea en el archivo_a.txt en la rama experimento
| * 117d623 (pruebas) Introduzco una línea en el archivo archivo_a.txt
|/
* 1a9f1c2 Añado el segundo archivo
* eff327b Añado el primer archivo
```

Vemos que las ramas "*master*" e "*intento*" apuntan al mismo commit.

Si nos cambiamos a la rama "master"

```
git checkout master
```

Y vemos el contenido del archivo "archivo_a.txt"

```
cat archivo_a.txt
```

```
Experimento añadiendo una nueva línea al archivo_a.txt en la rama experimento
```

vemos que los cambios introducidos en la rama "*experimento*", renombrada a "*intento*" en el apartado 5.7, han sido llevados a la rama "*master*".

Vamos a realizar un cambio en la rama "*master*" y otro en la rama "*intento*". Empecemos por la rama "*master*".

```
git checkout master
echo "Añado una segunda línea al archivo_a.txt en la rama master" >> archivo_a.txt
git commit -am "Añado una segunda línea al archivo_a.txt en la rama master"
```

```
[master 9229a2f] Añado una segunda línea al archivo_a.txt en la rama master
 1 file changed, 1 insertion(+)
```

Ahora hacemos lo mismo en la rama "*intento*", pero en un archivo distinto.

```
git checkout intento
echo "Añado una primera línea al archivo_b.txt en la rama intento"
>> archivo_b.txt
git commit -am "Añado una primera línea al archivo_b.txt en la rama
intento"
```

```
[intento 41077ec] Añado una primera línea al archivo_b.txt en la rama intento
 1 file changed, 1 insertion(+)
```

Si ejecutamos

```
git log --oneline --graph --all --decorate
```

```
* 41077ec (HEAD, intento) Añado una primera línea al archivo_b.txt en la rama
| * 9229a2f (master) Añado una segunda línea al archivo_a.txt en la rama
|/
* 9618f2a Experimentando con una nueva línea en el archivo_a.txt en la rama
| * 117d623 (pruebas) Introduzco una línea en el archivo archivo_a.txt
|/
* 1a9f1c2 Añado el segundo archivo
* eff327b Añado el primer archivo
```

Vemos que el último commit de las ramas "master" e "intento" tienen el mismo commit antecesor, por lo que ahora al fusionar las ramas no vamos a poder realizar un *fast-forward*, ya que las dos ramas han seguido caminos distintos a partir de un mismo commit.

Vamos a integrar los cambios de la rama "intento" en la rama "master". Para ello nos situamos en la rama "master" y realizamos el merge.

```
git checkout master
git merge intento
```

Se abre el editor que tenemos configurado por defecto por si queremos cambiar el mensaje que presenta por defecto, ya que va a crear un *commit de fusión* de forma automática con la fusión.

```
Merge branch 'intento'
```

```
# Please enter a commit message to explain why this merge is necessary,
# especially if it merges an updated upstream into a topic branch.
#
# Lines starting with '#' will be ignored, and an empty message aborts
# the commit.
```

Dejamos el mensaje tal y como está, guardamos y salimos del editor de texto para completar la fusión de las ramas, junto con el nuevo commit.

```
Merge made by the 'recursive' strategy.
 archivo_b.txt |    1 +
 1 file changed, 1 insertion(+)
```

Vemos que ahora indica que se ha llevado a cabo siguiendo una estrategia recursiva; es decir, Git ha buscado hacia atrás el commit común, ha visto qué cambios se habían introducido en la rama "*intento*" y los ha aplicado a la rama "*master*".

Si vemos el contenido del archivo "*archivo_b.txt*" en la rama "*master*"

```
git checkout master
cat archivo_b.txt
```

```
Añado una primera línea al archivo_b.txt en la rama intento
```

Vemos que el cambio que hemos introducido en la rama "*intento*" se ha trasladado a la rama "*master*".

Si ejecutamos

```
git log --oneline --graph --all --decorate
```

Vemos que se ha creado un commit de fusión, que aparece con el mensaje "Merge branch 'intento'", mensaje que hemos indicado en la fusión.

```
*   ef17121 (HEAD, master) Merge branch 'intento'
|\
| * 41077ec (intento) Añado una primera línea al archivo_b.txt en la rama
* | 9229a2f Añado una segunda línea al archivo_a.txt en la rama master
|/
* 9618f2a Experimentando con una nueva línea en el archivo_a.txt en la rama
| * 117d623 (pruebas) Introduzco una línea en el archivo archivo_a.txt
|/
* 1a9f1c2 Añado el segundo archivo
* eff327b Añado el primer archivo
```

5.10 Conflictos en la fusión entre ramas

Ahora vamos a tratar de llevar los cambios de la rama "pruebas" a la rama "master". Antes de empezar veamos el contenido del archivo "*archivo_a.txt*" en cada una de las ramas. Empezamos por la rama "*master*".

```
git checkout master
cat archivo_a.txt
```

```
Experimento añadiendo una nueva línea al archivo_a.txt en la rama experimento
Añado una segunda línea al archivo_a.txt en la rama master
```

A continuación hacemos lo mismo en la rama "pruebas".

```
git checkout pruebas
cat archivo_a.txt
```

```
Inserto una línea en el archivo_a.txt
```

Vemos que el contenido de la primera línea es diferente, por lo que puede que tengamos problemas al llevar el contenido de una rama a la otra. Vamos a probarlo, tratando de llevar los cambios de la rama "*pruebas*" a la rama "*master*".

```
git checkout master
git merge pruebas
```

```
Auto-merging archivo_a.txt
CONFLICT (content): Merge conflict in archivo_a.txt
Automatic merge failed; fix conflicts and then commit the result.
```

Vemos que la fusión falla, indicándonos que hay un conflicto de fusión en el archivo "*archivo_a.txt*".

Si vemos el estado del repositorio

```
git status
```

```
# On branch master
# Unmerged paths:
#   (use "git add/rm <file>..." as appropriate to mark resolution)
#
```

```
#  both modified:      archivo_a.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Nos indica que hay rutas que no están fusionadas y que no se han añadido cambios al commit.

Si vemos el contenido del archivo en conflicto

```
cat archivo_a.txt
```

```
<<<<<<< HEAD
Experimento añadiendo una nueva línea al archivo_a.txt en la rama experimento
Añado una segunda línea al archivo_a.txt en la rama master
=======
Inserto una línea en el archivo_a.txt
>>>>>>> pruebas
```

En este archivo ahora aparecen las dos partes en conflicto.

Las opciones que tenemos son:

- Abortar la fusión.
- Resolver el problema de forma manual y luego finalizar la fusión.
- Utilizar herramientas de fusión.

Vamos a ver las dos primeras.

Para abortar la fusión solo tenemos que ejecutar el comando

```
git merge --abort
```

Y todo vuelve al estado anterior al inicio de la fusión.

Si ejecutamos

```
git status
```

Vemos que ahora tenemos un directorio de trabajo limpio, sin cambios que llevar al repositorio.

```
# On branch master
nothing to commit (working directory clean)
```

Si vemos el contenido del archivo "archivo_a.txt"

```
cat archivo_a.txt
```

```
Experimento añadiendo una nueva línea al archivo_a.txt en la rama experimento
Añado una segunda línea al archivo_a.txt en la rama master
```

Vemos que ninguno de los cambios de la rama "*pruebas*" ha sido llevado a este archivo en la rama "*master*".

A continuación vamos a resolver el problema de forma manual. Para ello vamos a volver a ejecutar la fusión que nos llevó al problema

```
git checkout master
git merge pruebas
```

```
Auto-merging archivo_a.txt
CONFLICT (content): Merge conflict in archivo_a.txt
Automatic merge failed; fix conflicts and then commit the result.
```

Volvemos a ver el contenido del archivo en conflicto

```
cat archivo_a.txt
```

```
<<<<<<< HEAD
Experimento añadiendo una nueva línea al archivo_a.txt en la rama experimento
Añado una segunda línea al archivo_a.txt en la rama master
=======
Inserto una línea en el archivo_a.txt
>>>>>>> pruebas
```

Vemos que la parte que está en conflicto en la rama activa, la "*master*", está delimitada entre las marcas

```
<<<<<<< HEAD
```

Y

```
=======
```

Mientras que la parte que esta en conflicto en la rama desde la que tratamos de llevar los cambios, "*pruebas*", está delimitada entre las marcas

```
=======
```

Y

```
>>>>>>> pruebas
```

Lo que tenemos que hacer ahora es editar el archivo, eliminando las marcas que delimitan el problema. Lo normal es hacerlo con un editor de texto, pero en este caso lo hacemos con el comando "*echo*".

```
echo "Resuelvo el conflicto en esta primera línea del archivo_a.txt"
> archivo_a.txt

echo "También modifico la segunda línea del archivo_a.txt" >>
archivo_a.txt
```

La primera orden elimina el contenido anterior que hay en el archivo y la segunda añade esa línea al archivo, con lo que el contenido del archivo

```
cat archivo_a.txt
```

```
Resuelvo el conflicto en esta primera línea del archivo_a.txt
También modifico la segunda línea del archivo_a.txt
```

El contenido del archivo tendrá que ser el resultado de resolver de forma manual los conflictos, y siempre eliminando los indicadores de inicio y de fin de conflicto.

Si vemos el estado del repositorio

```
git status
```

```
# On branch master
# Unmerged paths:
#   (use "git add/rm <file>..." as appropriate to mark resolution)
#
#  both modified:      archivo_a.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Vemos que el conflicto sigue, pero que nos indica que usemos el comando "git add <archivo>" para marcar la resolución del conflicto.

Ejecutamos

```
git add archivo_a.txt
```

Si ahora vemos el estado del repositorio

```
git status
```

```
# On branch master
# Changes to be committed:
#
#  modified:   archivo_a.txt
#
```

Vemos que los cambios ya están listos para realizar el commit.

```
git commit -m "Resuelto el conflicto en la línea 1 en el
archivo_a.txt"
```

```
[master 70b7bef] Resuelto el conflicto en la línea 1 en el archivo_a.txt
```

Si vemos el histórico de commits

```
git log --oneline --graph --all --decorate -6
```

Podemos ver que la rama "pruebas" se ha integrado en la rama "master".

```
*   70b7bef (HEAD, master) Resuelto el conflicto en la línea 1 en el
|\
| * 117d623 (pruebas) Introduzco una línea en el archivo archivo_a.txt
* |   ef17121 Merge branch 'intento'
|\ \
| * | 41077ec (intento) Añado una primera línea al archivo_b.txt en la rama
* | | 9229a2f Añado una segunda línea al archivo_a.txt en la rama master
|/ /
* | 9618f2a Experimentando con una nueva línea en el archivo_a.txt en la rama
|/
```

5.11 Rebase

Hemos visto que una forma que tenemos de integrar los cambios de una rama en la otra es mediante la fusión, usando el comando "git merge".

Veámoslo con un ejemplo. Inicializamos un proyecto.

```
mkdir ~/proyectos/proyecto_rebase
cd ~/proyectos/proyecto_rebase
git init
```

Añadimos un archivo "archivo_1.txt" y realizamos el commit.

```
touch archivo_1.txt
git add .
git commit -m "Commit inicial"
```

Creamos una rama, nos cambiamos a ella, añadimos una línea al archivo “archivo_1.txt” y realizamos el commit.

```
git checkout -b rama_1
echo "Línea 1" >> archivo_1.txt
git commit -am "Añado una línea"
```

Volvemos a la rama master, añadimos un archivo “archivo_2.txt” y realizamos el commit.

```
git checkout master
echo "Línea 1" >> archivo_2.txt
git add .
git commit -m "Añado un nuevo archivo"
```

A continuación llevamos los cambios de la rama_1 a la rama master

```
git merge rama_1
```

Si vemos la línea temporal de commits podemos ver los que pertenecen a la rama master, los que pertenecen a la rama rama_1 y el commit de fusión.

```
git log --oneline --graph --all --decorate
```

```
*   05641c0 (HEAD, master) Merge branch 'rama_1'
|\
| * 5c87e40 (rama_1) Añado una línea
* | 0b95a9c Añado un nuevo archivo
|/
* d028e71 Commit inicial
```

Pero tenemos otra forma de fusionar estos cambios, que es obteniendo todos los parches que introducen los commits de la rama rama_1 y aplicándolos a la rama master en un nuevo commit. Esto lo hacemos con el comando rebase.

Veámoslo con un ejemplo similar al anterior. Inicializamos un proyecto.

```
mkdir ~/proyectos/proyecto_rebase_2
cd ~/proyectos/proyecto_rebase_2
```

```
git init
```

Añadimos un archivo “archivo_1.txt” y realizamos el commit.

```
touch archivo_1.txt
git add .
git commit -m "Commit inicial"
```

Creamos una rama, nos cambiamos a ella, añadimos una línea al archivo “archivo_1.txt” y realizamos el commit.

```
git checkout -b rama_1
echo "Línea 1" >> archivo_1.txt
git commit -am "Añado una línea"
```

Volvemos a la rama master, añadimos un archivo “archivo_2.txt” y realizamos el commit.

```
git checkout master
echo "Línea 1" >> archivo_2.txt
git add .
git commit -m "Añado un nuevo archivo"
git log --oneline --graph --all --decorate
```

Si vemos el estado de los commits, ahora mismo tenemos dos commits en 2 ramas que parten de un mismo commit común.

```
* ea41525 (HEAD, master) Añado un nuevo archivo
| * 5f4366f (rama_1) Añado una línea
|/
* 308d7d6 Commit inicial
```

Si quermos llevar los cambios de la rama_1 a la rama master aplicando el parche a la rama master, lo primero que hacemos es situarnos en la rama donde tenemos los parches.

```
git checkout rama_1
```

Y ejecutamos el comando pasando como parámetro la rama sobre la que queremos aplicar el rebase.

```
git rebase master
```

Lo que acaba de hacer el comando es buscar el antecesor común a las dos ramas, obtener las diferencias introducidas en cada commit por la rama activa (rama_1 en este caso), guarda de forma temporal esas diferencias o parches, lleva el puntero de la rama activa al último commit de la rama sobre la que queremos aplicar el rebase y aplica los parches.

```
git log --oneline --graph --all --decorate
```

Si vemos el gráfico vemos como ahora tenemos un historial lineal

```
* 8b6f8f6 (HEAD, rama_1) Añado una línea
* ea41525 (master) Añado un nuevo archivo
* 308d7d6 Commit inicial
```

Ahora solo nos queda fusionar, mediante un fast-forward, los cambios de la rama_1 en la rama master.

```
git checkout master
git merge rama_1
```

```
Updating ea41525..8b6f8f6
Fast-forward
 archivo_1.txt | 1 +
 1 file changed, 1 insertion(+)
```

Si vemos el gráfico vemos como ahora tenemos los dos punteros de las ramas apuntando al mismo commit.

```
git log --oneline --graph --all --decorate
```

```
* 8b6f8f6 (HEAD, rama_1, master) Añado una línea
* ea41525 Añado un nuevo archivo
* 308d7d6 Commit inicial
```

5.12 Stash

A veces estamos trabajando en una rama determinada y necesitamos cambiar a otra rama sin tener que realizar un commit, pero queremos guardar los cambios que hemos realizado en esa rama. Para esto utilizaremos el comando stash.

Lo que va a hacer el comando es guardar los cambios que hemos realizado desde el último commit en una zona temporal común a todas las ramas y deja nuestro directorio de trabajo limpio.

Vamos a verlo con un ejemplo.

Para ello creamos un directorio e inicializamos un repositorio.

```
mkdir ~/proyectos/proyecto_stash
cd ~/proyectos/proyecto_stash
```

```
git init
git branch rama_1
```

Si creamos una rama si un comit de partida nos encontramos con este error.

```
fatal: Not a valid object name: 'master'.
```

Por lo tanto, vamos a crear un primer commit para el proyecto y un par de cambios más sin guardar en el repositorio.

```
echo "Primera línea del archivo_1.txt" >> archivo_1.txt
git add .
git commit -m "Commit inicial"
echo "Segunda línea del archivo_1.txt" >> archivo_1.txt
echo "Primera línea del archivo_2.txt" >> archivo_2.txt
ls -la
```

```
archivo_1.txt
archivo_2.txt
.git
```

Ahora recibimos una interrupción y tenemos que empezar a trabajar en un cambio que tiene que tomar como base el último commit, sin las modificaciones que acabamos de introducir.

```
git status
```

```
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

   modified:   archivo_1.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)

   archivo_2.txt
```

```
no changes added to commit (use "git add" and/or "git commit -a")
```

Tenemos un archivo con cambios y otro sin estar bajo seguimiento. Para poder utilizarlo con los comandos de Git tenemos que ponerlo bajo seguimiento.

```
git add archivo_2.txt
git status
```

```
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

   new file:   archivo_2.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

   modified:   archivo_1.txt
```

Ahora los dos archivos se encuentran bajo seguimiento, aunque tienen cambios respecto al último commit.

Si vemos su contenido

```
cat archivo_1.txt
```

```
Primera línea del archivo_1.txt
Segunda línea del archivo_1.txt
```

```
cat archivo_2.txt
```

```
Primera línea del archivo_2.txt
```

Ahora vamos a almacenar estos cambios en el área de stash.

```
git stash
```

```
Saved working directory and index state WIP on master: 08f2d94 Commit inicial
HEAD is now at 08f2d94 Commit inicial
```

Si vemos el contenido del directorio de trabajo

```
ls -a -1
```

```
archivo_1.txt
.git
```

Acaba de desaparecer el archivo_2.txt, ya que se incorporó después del último commit.

Si vemos el contenido de archivo_1.txt

```
cat archivo_1.txt
```

```
Primera línea del archivo_1.txt
```

Vemos que también han desaparecido los cambios realizados desde el último commit.

Si tratamos de volver a ejecutar el stath

```
git stash
```

Vemos que no hay cambios que poder guardar.

```
No local changes to save
```

Lo siguiente que podemos ver son la lista de cambios almacenados en la área de stash.

```
git stash list
```

```
stash@{0}: WIP on master: 949c7b0 Commit inicial
```

que mostraría una lista si tuviéramos más elementos.

Si queremos ver los archivos afectados en los cambios que se almacenan en un elemento del área de stash.

```
git stash show stash@{0}
```

```
archivo_1.txt | 1 +
 archivo_2.txt | 1 +
 2 files changed, 2 insertions(+)
```

Si queremos ver los cambios introducidos ejecutamos

```
git stash show stash@{0} -p
```

```
diff --git a/archivo_1.txt b/archivo_1.txt
index 34c5d11..19265b1 100644
--- a/archivo_1.txt
+++ b/archivo_1.txt
@@ -1 +1,2 @@
 Primera línea del archivo_1.txt
+Segunda línea del archivo_1.txt
diff --git a/archivo_2.txt b/archivo_2.txt
new file mode 100644
index 0000000..b054e56
--- /dev/null
+++ b/archivo_2.txt
@@ -0,0 +1 @@
+Primera línea del archivo_2.txt
```

Lo siguiente que vamos a hacer es incorporar estos cambios almacenados en la zona de stash en una rama concreta.

Creamos una rama desde la rama master, nos cambiamos a ella y vemos su contenido.

```
git checkout -b rama_1
ls -a -1
```

```
archivo_1.txt
.git
```

Y lo siguiente es incorporar estos cambios almacenados en la zona de stash a la rama activa.

```
git stash apply stash@{0}
```

Al ser el primer elemento podríamos ejecutar

```
git stash apply
```

```
On branch rama_1
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

	new file:   archivo_2.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

	modified:   archivo_1.txt
```

Vemos que hemos recuperado en la rama rama_1 los cambios almacenados previamente en el área de stash desde la rama master.

6 Historial

Clono el proyecto https://github.com/github/gitignore

```
git clone https://github.com/github/gitignore.git
```

```
Clonar en «gitignore»...
remote: Counting objects: 4949, done.
remote: Compressing objects: 100% (32/32), done.
remote: Total 4949 (delta 14), reused 0 (delta 0), pack-reused 4916
Receiving objects: 100% (4949/4949), 1.09 MiB | 513.00 KiB/s, done.
Resolving deltas: 100% (2408/2408), done.
Checking connectivity... hecho.
```

```
# Último commit
git log -n 1
```

```
# Últimos 2 commits
git log -2
```

```
# Commits de "Gary Smith"
git log --author="Gary Smith"
```

```
# Commits de "Gary Smith" (1 por línea)
git log --oneline --author="Gary Smith"
```

```
# Commits desde el 1 de octubre del 2015
git log --since="2015-10-01" --oneline
```

```
# Commits desde el 1 de octubre hasta el 10 de octubre del 2015
```

```
git log --since="2015-10-01" --until="2015-10-10" --oneline
```

```
# Commits desde hace 3 semanas hasta hace 2 semanas
git log --since="3 weeks ago" --until="2 weeks ago" --oneline
```

```
# Commits desde hace 3 semanas hasta hace 2 semanas
git log --since=3.weeks --until=2.weeks --oneline
```

```
# Commits con la palabra “Emacs”
git log --grep="Emacs" --oneline
```

```
# Commits entre 2 commits
git log 4892e96..d06a6e2 --oneline
```

```
# Commits que afectan a un archivo determinado
git log --oneline D.gitignore
```

```
# Commits con más inforamción (patch)
git log -p D.gitignore
```

```
# Commits con formato grafo de todas las ramas
git log --oneline --graph --all --decorate
```

```
# Información del número de modificaciones en cada commit
git log --stat --oneline
```

7 Cambios en el proyecto

Para comprobar las diferencias entre dos commits usamos el commando

```
git diff
```

con una serie de parámetros.

```
cd ~
cd proyectos
mkdir cambios
cd cambios
git init
echo "Línea 1 archivo_a.txt" >> archivo_a.txt
git add .
git commit -m "Línea 1 al archivo_a.txt"
echo "Línea 2 archivo_a.txt" >> archivo_a.txt
echo "Línea 1 archivo_b.txt" >> archivo_b.txt
git add .
git commit -m "Línea 1 al archivo_b.txt y línea 2 al archivo_a.txt"
echo "Línea 3 archivo_a.txt" >> archivo_a.txt
echo "Línea 2 archivo_b.txt" >> archivo_b.txt
echo "Línea 1 archivo_cd c.txt" >> archivo_c.txt
```

```
git status
```

```
En la rama master
Cambios no preparados para el commit:
  (use «git add <archivo>...» para actualizar lo que se ejecutará)
  (use «git checkout -- <archivo>...« para descartar cambios en le directorio de trabajo)

   modified:   archivo_a.txt
   modified:   archivo_b.txt

Archivos sin seguimiento:
  (use «git add <archivo>...» para incluir lo que se ha de ejecutar)
```

```
    archivo_c.txt

no hay cambios agregados al commit (use «git add» o «git commit -a»)
```

```
git diff
```

```
diff --git a/archivo_a.txt b/archivo_a.txt
index 60d32d5..3cf25e9 100644
--- a/archivo_a.txt
+++ b/archivo_a.txt
@@ -1,2 +1,3 @@
 Línea 1 archivo_a.txt
 Línea 2 archivo_a.txt
+Línea 3 archivo_a.txt
diff --git a/archivo_b.txt b/archivo_b.txt
index 6342c05..d56f9a4 100644
--- a/archivo_b.txt
+++ b/archivo_b.txt
@@ -1 +1,2 @@
 Línea 1 archivo_b.txt
+Línea 2 archivo_b.txt
```

```
git add .
git status
```

```
En la rama master
Cambios para hacer commit:
  (use «git reset HEAD <archivo>...«para eliminar stage)

    modified:   archivo_a.txt
    modified:   archivo_b.txt
    new file:   archivo_c.txt
```

```
git diff
```

```
```

```
git diff --staged # git diff --cached
```

```
diff --git a/archivo_a.txt b/archivo_a.txt
index 60d32d5..3cf25e9 100644
--- a/archivo_a.txt
+++ b/archivo_a.txt
@@ -1,2 +1,3 @@
 Línea 1 archivo_a.txt
 Línea 2 archivo_a.txt
+Línea 3 archivo_a.txt
diff --git a/archivo_b.txt b/archivo_b.txt
index 6342c05..d56f9a4 100644
--- a/archivo_b.txt
+++ b/archivo_b.txt
@@ -1 +1,2 @@
 Línea 1 archivo_b.txt
+Línea 2 archivo_b.txt
diff --git a/archivo_c.txt b/archivo_c.txt
new file mode 100644
index 0000000..d4a4db0
--- /dev/null
+++ b/archivo_c.txt
@@ -0,0 +1 @@
+Línea 1 archivo_c.txt
```

```
git commit -m "Línea 1 archivo_c.txt, línea 2 archivo_b.txt, línea 3 archivo_a.txt"
git diff
```

```
```

```
git diff --staged
```

```

```

```
git log --oneline
```

```
d1234b5 Línea 1 archivo_c.txt, línea 2 archivo_b.txt, línea 3 archivo_a.txt
ad4027e Línea 1 al archivo_b.txt y línea 2 al archivo_a.txt
f6295ca Línea 1 al archivo_a.txt
```

```
git diff ad4027e
```

```
diff --git a/archivo_a.txt b/archivo_a.txt
index 60d32d5..3cf25e9 100644
--- a/archivo_a.txt
+++ b/archivo_a.txt
@@ -1,2 +1,3 @@
 Línea 1 archivo_a.txt
 Línea 2 archivo_a.txt
+Línea 3 archivo_a.txt
diff --git a/archivo_b.txt b/archivo_b.txt
index 6342c05..d56f9a4 100644
--- a/archivo_b.txt
+++ b/archivo_b.txt
@@ -1 +1,2 @@
 Línea 1 archivo_b.txt
+Línea 2 archivo_b.txt
diff --git a/archivo_c.txt b/archivo_c.txt
new file mode 100644
index 0000000..d4a4db0
--- /dev/null
+++ b/archivo_c.txt
@@ -0,0 +1 @@
```

```
+Línea 1 archivo_c.txt
```

```
git diff ad4027e..d1234b5
git diff ad4027e..HEAD
```

```
git diff d1234b5..ad4027e
```

```
diff --git a/archivo_a.txt b/archivo_a.txt
index 3cf25e9..60d32d5 100644
--- a/archivo_a.txt
+++ b/archivo_a.txt
@@ -1,3 +1,2 @@
 Línea 1 archivo_a.txt
 Línea 2 archivo_a.txt
-Línea 3 archivo_a.txt
diff --git a/archivo_b.txt b/archivo_b.txt
index d56f9a4..6342c05 100644
--- a/archivo_b.txt
+++ b/archivo_b.txt
@@ -1,2 +1 @@
 Línea 1 archivo_b.txt
-Línea 2 archivo_b.txt
diff --git a/archivo_c.txt b/archivo_c.txt
deleted file mode 100644
index d4a4db0..0000000
--- a/archivo_c.txt
+++ /dev/null
@@ -1 +0,0 @@
-Línea 1 archivo_c.txt
```

```
git diff --stat ad4027e..HEAD
```

```
 archivo_a.txt | 1 +
```

```
archivo_b.txt | 1 +
archivo_c.txt | 1 +
3 files changed, 3 insertions(+)
```

```
git diff f6295ca..ad4027e archivo_b.txt
```

```
diff --git a/archivo_b.txt b/archivo_b.txt
new file mode 100644
index 0000000..6342c05
--- /dev/null
+++ b/archivo_b.txt
@@ -0,0 +1 @@
+Línea 1 archivo_b.txt
```

8 Etiquetas (tags)

```
cd ~/proyectos/
git clone https://github.com/jquery/jquery.git
cd jquery
git tag
```

```
git tag -l '2.1.0*'
```

```
2.1.0
2.1.0-beta1
2.1.0-beta2
2.1.0-beta3
2.1.0-rc1
```

```
git tag -a 3.0.1 -m "Tag de prueba"
```

```
git tag -s 3.0.0.1 f931786 -m "Tag de prueba 2"
```

```
gpg: /home/usuario/.gnupg: directorio creado
gpg: creado un nuevo archivo de configuración `/home/usuario/.gnupg/gpg.conf'
gpg: AVISO: las opciones en `/home/usuario/.gnupg/gpg.conf' no están aún
activas en esta ejecución
gpg: anillo «/home/usuario/.gnupg/secring.gpg» creado
gpg: anillo «/home/usuario/.gnupg/pubring.gpg» creado
gpg: omitido «Jesús Amieiro <amieiro@gmail.com>»: clave secreta no disponible
gpg: signing failed: clave secreta no disponible
error: gpg falló al firmar los datos
error: unable to sign the tag
```

```
git tag 3.0.0.1 f931786 -m "Tag de prueba 2"
```

```
git status
```

```
En la rama master
Su rama está actualizada con «origin/master».
nothing to commit, working directory clean
```

```
git tag -d 3.0.0.1
```

```
Deleted tag '3.0.0.1' (was efd4331)
```

9 Alias

```
git config --global alias.co checkout
git config --global alias.br branch
git config --global alias.ci commit
git config --global alias.st status
git config --global alias.unstage 'reset HEAD --'
git config --global alias.last 'log -1 HEAD'
```

```
 alias gs='git status '              # Show the working tree status
 alias gsh='git show '               # Show various types of objects
 alias ga='git add '                 # Add file contents to the
index
 alias ga.='git add . '              # Add all file contents to the
index
 alias gl="git log --graph --pretty=format:'%an: %s - %Cred%h%Creset
%C(yellow)%d%Creset %Cgreen(%cr)%Creset' --abbrev-commit
--date=relative -10"
 alias gl10='git log --oneline -10 ' # Show the last 10 commits, 1
per line
 alias gb='git branch '              # List local branches
 alias gba='git branch -a '          # List all branches
 alias gc='git commit '              # Record changes to the
repository
 alias gd='git diff '                # Changes in the working tree
not yet staged for the next commit.
 alias gd1='git diff HEAD^ HEAD '    # Compare the version before
the last commit and the last commit.
 alias gd2='git diff HEAD^^ HEAD '   # Compare the version before
the penultimate commit and the last commit.
 alias gds='git diff --staged '      # Changes between the index and
your last commit
 alias go='git checkout '            # Checkout a branch
 alias gob='git checkout -b '        # Create a new branch and
checkout it
 alias gr='git remote '
 alias grv='git remote -v '          #
 alias gph='git push '               # Update remote refs along with
associated objects
 alias gpl='git pull '               #
```

10 SSH

```
mkdir ~/.ssh
cd ~/.ssh
ssh-keygen -t rsa -C "amieiro@gmail.com" -f ~/.ssh/id_rsa_amieiro
```

```
Generating public/private rsa key pair.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/usuario/.ssh/id_rsa_amieiro.
Your public key has been saved in /home/usuario/.ssh/id_rsa_amieiro.pub.
The key fingerprint is:
SHA256:HGgAOlCMUwVJ9VFOzp/TwZ1zUKJSwAhGfk6MAJOp8nI amieiro@gmail.com
The key's randomart image is:
+---[RSA 2048]----+
|.*OX=.=ooo... o..|
|+.=. * X. .o o + |
|oo    * O . + + .|
|o.   . = o + . o |
|..      S + .    |
|. E        .     |
| o               |
|                 |
|                 |
+----[SHA256]-----+
```

```
ls -1 -a
```

```
id_rsa_amieiro
id_rsa_amieiro.pub
known_hosts
```

```
cat id_rsa_amieiro.pub
```

```
ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAABAQCchjtV+dqFSAe3v9yuyo3DXFoJWxVSzEv9pCkZN9WuXIXjns
Q0rI59AuwtXe6PPaDb/J+mzWbcRLTfYfO7xMoOlzl+WLTLMvld2YjP+YQdYCXnO6GAv4qY5vs3ttXl
h3hVvv8j9eNyLLYIigQ03mnq8njwse4OB07X9OCisjpj8J+TUDxCA7+tK2XC+Vfc2gzD+
+0UpdpNzcthGMV8th/UhZqSRJeTjVQoJjIIxaV5sP0IUIFlSrbDtnHN7ILrYTTxEdayezlrI2oCw1C
JlPB1+JMTnTPyG7QPBgZPRyg+pnNqRcUUTMjRIfPCrY7xVJJK6C0KZzud3bk3/pIVMMCV
amieiro@gmail.com
```

GitHub → Settings → SSH keys → Add SSH Key

BitBucket → Manage account → SSH keys → Add Key

Pegar el contenido del archivo “id_rsa_amieiro.pub”.

```
ssh -T git@github.com
```

```
The authenticity of host 'github.com (192.30.252.128)' can't be established.

RSA key fingerprint is SHA256:nThbg6kXUpJWGl7E1IGOCspRomTxdCARLviKw6E5SY8.

Are you sure you want to continue connecting (yes/no)? yes

Warning: Permanently added 'github.com,192.30.252.128' (RSA) to the list of
known hosts.

Hi amieiro! You've successfully authenticated, but GitHub does not provide
shell access.
```

```
ssh -T git@bitbucket.com
```

```
The authenticity of host 'bitbucket.com (131.103.20.172)' can't be
established.

RSA key fingerprint is SHA256:zzXQOXSRBEiUtuE8AikJYKwbHaxvSc0ojez9YXaGp1A.

Are you sure you want to continue connecting (yes/no)? yes

Warning: Permanently added 'bitbucket.com,131.103.20.172' (RSA) to the list of
known hosts.

logged in as amieiro.


You can use git or hg to connect to Bitbucket. Shell access is disabled.
```

```
git clone git@bitbucket.org:amieiro/proyecto-compartido.git
proyecto-compartido-2
```

```
Clonar en «proyecto-compartido-2»...
remote: Counting objects: 16, done.
remote: Compressing objects: 100% (9/9), done.
remote: Total 16 (delta 1), reused 0 (delta 0)
Receiving objects: 100% (16/16), done.
Resolving deltas: 100% (1/1), done.
Checking connectivity... hecho.
```

```
git remote -v
```

```
origin   git@bitbucket.org:amieiro/proyecto-compartido.git (fetch)
origin   git@bitbucket.org:amieiro/proyecto-compartido.git (push)
```

```
git branch -a -v
```

```
* master                 5a30a8f Añado el archivo jquery-2.1.0.min.js
  remotes/origin/HEAD   -> origin/master
  remotes/origin/dev    5a30a8f Añado el archivo jquery-2.1.0.min.js
  remotes/origin/master 5a30a8f Añado el archivo jquery-2.1.0.min.js
```

```
git checkout -b dev origin/dev
```

```
Branch dev set up to track remote branch dev from origin.
Switched to a new branch 'dev'
```

```
git branch -a -v
```

```
* dev                    5a30a8f Añado el archivo jquery-2.1.0.min.js
  master                 5a30a8f Añado el archivo jquery-2.1.0.min.js
  remotes/origin/HEAD   -> origin/master
```

```
          remotes/origin/dev    5a30a8f Añado el archivo jquery-2.1.0.min.js
          remotes/origin/master 5a30a8f Añado el archivo jquery-2.1.0.min.js
```

11 Complementos de la consola

Bash-it https://github.com/Bash-it/bash-it

Instalación

```
git clone --depth=1 https://github.com/Bash-it/bash-it.git
~/.bash_it

~/.bash_it/install.sh

source ~/.bashrc
```

Edit your modified config (~/.bash_profile or ~/.bashrc) file in order to customize Bash it.

12 GUI

```
apt-get install git-gui
git gui &
```

Edito el archivo README.md

Rescan

Hago commit

```
gitk &
```

http://www.collab.net/products/giteye

https://www.sourcetreeapp.com/

13 Repositorios remotos

Para poder seguir los ejemplos de este capítulo es necesario disponer de dos cuentas en BitBucket u otro sistema similar, como GitHub.

Puedes crear una cuenta gratuita en la siguiente dirección https://bitbucket.org/account/signup/